Оглавление

Предисловие

Эта книга подробно рассказывает о современных методах построения и анализа алгоритмов. В ней подробно разобрано много конкретных алгоритмов; мы старались рассказать о них понятно, но не опуская деталей и не жертвуя строгостью изложения.

Алгоритмы записаны с виде *"псевдокода"* и прокомментированы в тексте; мы старались сделать описание алгоритма понятным людям с минимальным программистским опытом. Книга содержит более 260 рисунков, поясняющих работу различных алгоритмов. Мы обращаем особое внимание на эффективность рассматриваемых алгоритмов и приводим оценки времени их работы.

Мы старались написать учебник по построению алгоритмов и структур данных, который могли бы использовать преподаватели и студенты — от первокурсников до аспирантов. Книга может быть использована и для самообразования профессиональных программистов.

Преподавателям:

Мы старались сделать возможным использование книги на разных уровнях — от начального курса по программированию и структурам данных до аспирантского курса по эффективным алгоритмам. В ней гораздо больше материала, чем можно включить в семестровый курс, так что вы можете выбрать главы по вкусу.

Мы старались сделать главы достаточно независимыми. Каждая глава начинается с более простого материала; более трудные темы отнесены в разделы, помеченные звёздочкой и помещённые в конец главы. В лекциях для начинающих можно ограничиться несколькими первыми разделами выбранных вами глав, оставив подробное изучение остальных для более продвинутого курса.

Каждый раздел снабжён упражнениями (всего их более 900): каждая глава заканчивается задачами (всего более 120). Как правило, упражнения проверяют понимание изложенного материала (часть из них — устные вопросы, часть подходят для письменного домашнего задания). Задачи более развёрнуты; многие из них дополняют теоретический материал соответствующей главы и разбиты на части, соответствующие этапам доказательства или построения.

Звёздочкой отмечены более трудные упражнения и разделы; они предназначены скорее для старшекурсников и аспирантов. Разделы со звёздочкой часто требуют лучшей математической подготовки; упражнение со звёздочкой может также требовать дополнительных знаний или просто быть более трудным.

Студентам:

Мы надеемся, что книга доставит вам удовольствие и познакомит с методами построения алгоритмов. Мы старались писать подробно, понятно и интересно, напоминая по ходу дела необходимые сведения из математики. Подготовительные сведения обычно собраны в начальных разделах главы, которые можно бегло просмотреть, если вы уже знакомы с темой.

Книга эта велика, и на лекциях, скорее всего, будет разобрана лишь часть материала. Мы надеемся, что оставшаяся часть будет вам полезна если не сейчас, так в будущем, так что вы сохраните книгу в качестве справочника.

Что нужно знать, приступая к чтению? Мы рассчитываем, что вы

- имеете некоторый программистский опыт, и рекурсивные процедуры, массивы и списки вас не пугают;
- простые математические рассуждения (скажем, доказательства по индукции) вам также знакомы (кое-где понадобятся отдельные факты из курса математического анализа; в первой части больше ничего из математики не потребуется).

Программистам:

В книгу включены алгоритмы для самых разных задач и её можно использовать как справочник. Главы почти независимы, так что можно сразу выбрать интересующий вас материал.

Большинство обсуждаемых алгоритмов вполне могут быть использованы на практике, и мы уделяем должное внимание деталям реализации. Если алгоритм представляет скорее теоретический интерес, мы отмечаем это и обсуждаем альтернативные подходы.

Наш псевдокод легко перевести на любой язык программирования, если это понадобится. Надо только иметь в виду, что мы не включаем в алгоритмы системно-зависимые фрагменты (обработку ошибок и т. п.), чтобы не затемнять сути дела.

Ошибки

Книга такого объёма не может не содержать ошибок. Если вы обнаружили ошибку в английском оригинале книги, или у вас есть предложения по её исправлению, мы будем рады узнать об этом. Мы будем особенно рады новым упражнениям и задачам (но, пожалуйста, присылайте их с решениями). Почтовый адрес:

Introduction to Algorithms
MIT Labratory for Computer Science
545 Technology Square
Cambridge, Massachusetts 02139

Можно также получить список известных опечаток и сообщить о найденных ошибках с помощью электронной почты; чтобы получить инструкции, пошлите по адресу `algorithms@theory.lcs.mit.edu` письмо, содержащее `Subject: help` в заголовке. Извините, что мы не можем лично ответить на все письма.

[При переводе были учтены все исправления, имевшиеся на момент издания перевода (декабрь 1997), кроме того, исправлено несколько обнаруженных при переводе опечаток, но, возможно, возникли новые (в чём виноват научный редактор книги, А. Шень). Поэтому, обнаружив ошибку в русском тексте, не посылайте её сразу авторам: может быть, она возникла при переводе! Сообщите о ней сначала переводчикам, по адресу `algor@mccme.ru` или по почте (Москва, 121002, Большой Власьевский пер., 11, Московский центр непрерывного математического образования, издательство).]

Благодарности

Многие друзья и коллеги немало сделали для улучшения этой книги. Мы благодарим всех их за помощь и конструктивную критику.

Лаборатория информатики Массачусетского технологического института (Massachusetts Institute of Technology, Laboratory for Computer Science) была идеальным местом для работы над книгой. Наши коллеги по теоретической группе этой лаборатории были особенно терпимы и любезно соглашались просматривать главы книги. Мы хотели бы особенно поблагодарить следующих из них: Baruch Awerbuch, Shafi Goldwasser, Leo Guibas, Tom Leighton, Albert Meyer, David Shmoys, Eva Tardos. Компьютеры, на которых готовилась книга (трёх типов: Microvax, Apple Macintosh, Sun Sparcstation) поддерживали William Ang, Sally Bemus, Ray Hirschfeld и Mark Reinhold; они же перекомпилировали TEX, когда наши файлы перестали помещаться в его стандартную версию. Компания Thinking Machines поддерживала Чарльза Лейзерсона в

период его работы в этой компании.

Многие наши коллеги использовали предварительные варианты этой книги в своих лекционных курсах, и предложили различные улучшения. Мы хотели бы особенно поблагодарить следующих наших коллег: Richard Beigel (Yale), Andrew Goldberg (Stanford), Joan Lucas (Rutgers), Mark Overmars (Utrecht), Alan Sherman (Tufts, Maryland), Diane Souvaine (Rutgers).

При чтении лекций по материалам этой книги нам помогали наши коллеги, которые внесли много улучшений. Мы особенно признательны: Alan Baratz, Bonnie Berger, Aditi Dhagat, Burt Kaliski, Arthur Lent, Andrew Moulton, Marios Papaefthymiou, Cindy Phillips, Mark Reinhold, Phil Rogaway, Flavio Rose, Arie Rudich, Alan Sherman, Cliff Stein, Susmita Sur, Gregory Troxel, Margaret Tuttle.

Многие люди помогли нам в работе над книгой в разных отношениях: работа в библиотеке (Denise Sergent), гостеприимство в читальном зале (Maria Sensale), доступ к личной библиотеке (Albert Meyer), проверка упражнений и придумывание новых (Shlomo Kipnis, Bill Niehaus, David Wilson), составление индекса (Marios Papaefthymiou, Gregory Troxel), техническая помощь (Inna Radzihovsky, Denise Sergent, Gayle Sherman, и особенно Be Hubbard).

Многие ошибки обнаружили наши студенты, особенно Bobby Blumofe, Bonnie Eisenberg, Raymond Johnson, John Keen, Richard Lethin, Mark Lillibridge, John Pesaris, Steve Ponzio, Margaret Tuttle.

Многие наши коллеги сообщили нам полезную информацию о конкретных алгоритмах, а также критически прочитали отдельные главы книги; в их числе Bill Aiello, Alok Aggrawal, Eric Bach, Vašek Chvátal, Richard Cole, Johan Hastad, Alex Ishii, David Johnson, Joe Kilian, Dina Kravets, Bruce Maggs, Jim Orlin, James Park, Thane Plambeck, Herschel Safer, Jeff Shallit, Cliff Stein, Gil Strang, Bob Tarjan, Paul Wang. Многие из них предложили нам задачи для нашей книги, среди них Andrew Goldberg, Danny Sleator, Umesh Vazirani.

Английский оригинал книги был подготовлен с помощью LaTeX (макропакет для системы TeX). Рисунки делались на компьютере Apple Macintosh с помощью программы Mac Draw II; мы благодарны за оперативную техническую поддержку в этой области (Joanna Terry, Claris Corporation; Michael Mahoney, Advanced Computer Graphics). Индекс был подготовлен с помощью программы Windex, написанной авторами. Список литературы готовился с помощью программы BibTeX. Оригинал-макет английского издания был подготовлен в Американском математическом обществе с помощью фотонаборной машины фирмы Autologic; мы признательны за помощь в этом (Ralph Youngen, Американское математическое общество). Макет разработали: Rebecca Daw, Amy Henderson (реализация макета для системы LaTeX), Jeannet

Leendertse (обложка).

Авторы получили большое удовольствие от сотрудничества с издательствами MIT Press (Frank Sallow, Terry Ehling, Larry Cohen, Lorrie Lejeune) и McGraw-Hill (David Shapiro) и благодарны им за поддержку и терпение, а также замечательное редактирование (Larry Cohen).

Наконец, авторы благодарят своих жён (Nicole Cormen, Lina Lue Leicerson, Gail Rivest) и детей (Ricky, William и Debby Leicerson; Alex и Christopher Rivest) за любовь и поддержку при работе над книгой (Alex Rivest также помог нам с *"парадоксом дней рождения на Марсе"* (раздел 6.6.1). Любовь, терпение и поддержка наших семей сделали эту книгу возможной; им она и посвящается.

Кембридж,
Массачусетс
март 1990 года

Томас Кормен (Thomas H. Cormen)
Чарльз Лейзерсон (Charles E. Leiserson)
Рональд Ривест (Ronald L. Rivest)

От переводчиков:

Мы признательны издательству MIT и авторам за разрешение перевести книгу, и за помощь при подготовке перевода (в том числе за предоставление английского текста книги и иллюстраций в электронной форме). Издание перевода стало возможно благодаря финансовой поддержке Российского фонда фундаментальных исследований (руководитель проекта В.А. Успенский).

В работе над переводом книги участвовала большая группа студентов, аспирантов и сотрудников Московского центра непрерывного математического образования, Независимого Московского университета и МГУ:

К. Белов, Ю. Боравлёв, Д. Ботин, В. Горелик, Д. Дерягин Ю. Калнишкан, А. Катанова, С. Львовский, А. Ромащенко, К. Сонин, К. Трушкин, М. Ушаков, А. Шень, В. Шувалов, М. Юдашкин (перевод)

А. Акимов, М. Вьюгин, Д. Дерягин, А. Евфимьевский, Ю. Калнишкан, А. Ромащенко, А. Чернов М. Ушаков, А. Шень (редактирование)

В. Радионов (вёрстка)

В.В. Ященко (редактор)

1 Введение

В этой главе мы разбираем основные понятия и методы, связанные с построением и анализом алгоритмов, на примере двух алгоритмов сортировки — простейшего алгоритма сортировки вставками и более эффективного алгоритма сортировки слиянием.

На этих примерах мы познакомимся с псевдокодом, на котором мы будем записывать алгоритмы, обозначениями для скорости роста функций, методом *"разделяй и властвуй"* построения алгоритмов, а также с другими понятиями, которые будут встречаться нам на протяжении всей книги.

1.1 Алгоритмы

Алгоритм (algorithm) — это формально описанная вычислительная процедура, получающая **исходные данные** (input), называемые также входом алгоритма или его аргументом, и выдающая **результат** вычислений на выход (output).

Алгоритмы строятся для решения тех или иных **вычислительных задач** (computational problems). Формулировка задачи описывает, каким требованиям должно удовлетворять решение задачи, а алгоритм, решающий эту задачу, находит объект, этим требованиям удовлетворяющий.

В этой главе мы рассматриваем **задачу сортировки** (sorting problem); помимо своей практической важности эта задача служит удобным примером для иллюстрации различных понятий и методов. Она описывается так:

Вход: Последовательность n чисел $\langle a_1, a_2, \ldots, a_n \rangle$.

Выход: Перестановка $\langle a'_1, a'_2, \ldots, a'_n \rangle$ исходной последовательности, для которой $a'_1 \leqslant a'_2 \leqslant \ldots \leqslant a'_n$.

Например, получив на вход $\langle 31, 41, 59, 26, 41, 58 \rangle$, алгоритм сортировки должен выдать на выход $\langle 26, 31, 41, 41, 58, 59 \rangle$.

Подлежащая сортировке последовательность называется **входом** (instance) задачи сортировки.

Многие алгоритмы используют сортировку в качестве промежуточного шага. Имеется много разных алгоритмов сортировки; выбор в конкретной ситуации зависит от длины сортируемой последовательности, от того, в какой степени она уже отсортирована, а также от типа имеющейся памяти (оперативная память, диски, магнитные ленты).

Алгоритм считают **правильным** (correct), если на любом допустимом (для данной задачи) входе он он заканчивает работу и выдает результат, удовлетворяющий требованиям задачи. В этом случае говорят, что алгоритм **решает** (solves) данную вычислительную задачу. Неправильный алгоритм может (для некоторого входа) вовсе не остановиться или дать неправильный результат. (Впрочем, это не делает алгоритм заведомо бесполезным — если ошибки достаточно редки. Подобная ситуация встретится нам в главе 33 при поиске больших простых чисел. Но это всё же скорее исключение, чем правило.)

Алгоритм может быть записан на русском или английском языке, в виде компьютерной программы или даже в машинных кодах — важно только, чтобы процедура вычислений была чётко описана.

Мы будем записывать алгоритмы с помощью **псевдокода** (pseudocode), который напомнит вам знакомые языки программирования (Си, Паскаль, Алгол). Разница в том, что иногда мы позволяем себе описать действия алгоритма *"своими словами"*, если так получается яснее. Кроме того, мы опускаем технологические подробности (обработку ошибок, скажем), которые необходимы в реальной программе, но могут заслонить существо дела.

Сортировка вставками

Сортировка вставками (insertion sort) удобна для сортировки коротких последовательностей. Именно таким способом обычно сортируют карты: держа в левой руке уже упорядоченные карты и взяв правой рукой очередную карту, мы вставляем её в нужное место, сравнивая с имеющимися и идя справа налево (см. рис. 1.1)

Запишем этот алгоритм в виде процедуры Insertion-Sort, параметром которой является массив $A[1\,..\,n]$ (последовательность длины n, подлежащая сортировке). Мы обозначаем число элементов в массиве A через $length[A]$. Последовательность сортируется *"на месте"* (in place), без дополнительной памяти (помимо массива мы используем лишь фиксированное число ячеек памяти). После выполнения процедуры Insertion-Sort массив A упорядочен по возрастанию.

Рисунок 1.1 Сортировка карт вставками

```
INSERTION-SORT($A$)
for $j \leftarrow 2$ to $length[A]$
    do $key \leftarrow A[j]$
       ▷ добавить $A[j]$ к отсортированной части $A[1\,..\,j-1]$.
       $i \leftarrow j-1$
       while $i > 0$ and $A[i] > key$
           do $A[i+1] \leftarrow A[i]$
              $i \leftarrow i-1$
       $A[i+1] \leftarrow key$
```

Рисунок 1.2 Работа процедуры INSERTION-SORT для входа $A = \langle 5, 2, 4, 6, 1, 3\rangle$. Позиция j показана кружком.

На рис. 1.2 показана работа алгоритма при $A = \langle 5, 2, 4, 6, 1, 3\rangle$. Индекс j указывает *"очередную карту"* (только что взятую со стола). Участок $A[1\,..\,j-1]$ составляют уже отсортированные карты (левая рука), а $A[j+1\,..\,n]$ — ещё не просмотренные. В цикле **for** индекс j пробегает массив слева направо. Мы берём элемент

$A[j]$ (строка 2 алгоритма) и сдвигаем идущие перед ним и большие его по величине элементы (начиная с $j-1$-го) вправо, освобождая место для взятого элемента. (строки 4–7). В строке 8 элемент $A[j]$ помещается в освобождённое место.

Псевдокод

Вот основные соглашения, которые мы будем использовать:

1. Отступ от левого поля указывает на уровень вложенности. Например, тело цикла **for** (строка 1) состоит из строк 2–8, а тело цикла **while** (строка 5) содержит строки 6–7, но не 8. Это же правило применяется и для **if-then-else**. Это делает излишним специальные команды типа **begin** и **end** для начала и конца блока. (В реальных языках программирования такое соглашение применяется редко, поскольку затрудняет чтение программ, переходящих со страницы на страницу.)

2. Циклы **while**, **for**, **repeat** и условные конструкции **if**, **then**, **else** имеют тот же смысл, что в Паскале.

3. Символ ▷ начинает комментарий (идущий до конца строки).

4. Одновременное присваивание $i \leftarrow j \leftarrow e$ (переменные i и j получают значение e) заменяет два присваивания $j \leftarrow e$ и $i \leftarrow j$ (в этом порядке).

5. Переменные (в данном случае i, j, key) локальны внутри процедуры (если не оговорено противное).

6. Индекс массива пишется в квадратных скобках: $A[i]$ есть i-й элемент в массиве A. Знак ".." выделяет часть массива: $A[1\,..\,j]$ обозначает участок массива A, включающий $A[1], A[2], \ldots, A[j]$.

7. Часто используются **объекты** (objects), состоящие из нескольких **полей** (fields), или, как говорят, имеющие несколько **атрибутов** (attributes). Значение поля записывается как *имя_поля*[*имя_объекта*]. Например, длина массива считается его атрибутом и обозначается *length*, так что длина массива A запишется как $length[A]$. Что обозначают квадратные скобки (элемент массива или поле объекта), будет ясно из контекста.

Переменная, обозначающая массив или объект, считается указателем на составляющие его данные. После присваивания $y \leftarrow x$ для любого поля f выполнено $f[y] = f[x]$. Более того, если мы теперь выполним оператор $f[x] \leftarrow 3$, то будет не только $f[x] = 3$, но и $f[y] = 3$, поскольку после $y \leftarrow x$ переменные x и y указывают на один и тот же объект.

Указатель может иметь специальное значение NIL, не указывающее ни на один объект.

8. Параметры передаются **по значению** (by value): вызванная процедура получает собственную копию параметров; изменение параметра внутри процедуры снаружи невидимо. При передаче объек-

тов копируется указатель на данные, составляющие этот объект, а сами поля объекта — нет. Например, если x — параметр процедуры, то присваивание $x \leftarrow y$, выполненное внутри процедуры, снаружи заметить нельзя, а присваивание $f[x] \leftarrow 3$ — можно.

Упражнения

1.1-1 Следуя образцу рис. 1.2, покажите, как работает INSERTION-SORT на входе $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

1.1-2 Измените процедуру INSERTION-SORT так, чтобы она сортировала числа в невозрастающем порядке (вместо неубывающего).

1.1-3 Рассмотрим следующую **задачу поиска**:

Вход: Последовательность n чисел $A = \langle a_1, a_2, \ldots, a_n \rangle$ и число v.

Выход: Индекс i, для которого $v = A[i]$, или специальное значение NIL, если v не встречается в A.

Напишите программу **линейного поиска** (linear search), который последовательно просматривает A в поисках v.

1.1-4 Даны два n-значных двоичных числа, записанных в виде n-элементных массивов A и B. Требуется поместить их сумму (в двоичной записи) в $(n+1)$-элементный массив C. Уточните постановку задачи и запишите соответствующую программу на псевдокоде.

1.2 Анализ алгоритмов

Рассматривая различные алгоритмы решения одной и той же задачи, полезно проанализировать, сколько вычислительных ресурсов они требуют (время выполнения, память), и выбрать наиболее эффективный. Конечно, надо договориться о том, какая модель вычислений используется. В этой книге в качестве модели по большей части используется обычная однопроцессорная **машина с произвольным доступом** (random-access machine, RAM), не предусматривающая параллельного выполнения операций. (Мы рассмотрим некоторые модели параллельных вычислений в последней части книги.)

Сортировка вставками: анализ

Время сортировки вставками зависит от размера сортируемого массива: чем больше массив, тем больше может потребоваться времени. Обычно изучают зависимости времени работы от размера входа. (Впрочем, для алгоритма сортировки вставками важен не только размер массива, но и порядок его элементов: если массив почти упорядочен, то времени требуется меньше.)

Как измерять **размер входа** (input size)? Это зависит от конкретной задачи. В одних случаях разумно считать число элементов на входе (сортировка, преобразование Фурье). В других более естественно считать общее число битов, необходимое для представления всех входных данных. Иногда размер входа измеряется не одним числом, а несколькими (например, число вершин и число рёбер графа).

Временем работы (running time) алгоритма мы называем число элементарных шагов, которые он выполняет — вопрос только в том, что считать элементарным шагом. Мы будем полагать, что одна строка псевдокода требует не более чем фиксированного числа операций (если только это не словесное описание какой-то сложной операции — типа *"отсортировать все точки по x-координате"*). Мы будем различать также **вызов** (call) процедуры (на который уходит фиксированное число операций) и её **исполнение** (execution), которое может быть долгим.

Итак, вернёмся к процедуре Insertion-Sort и отметим около каждой строки её стоимость (число операций) и число раз, которое эта строка исполняется. Для каждого j от 2 до n (здесь $n = length[A]$ — размер массива) подсчитаем, сколько раз будет исполнена строка 5, и обозначим это число через t_j. (Заметим, что строки внутри цикла выполняются на один раз меньше, чем проверка, поскольку последняя проверка выводит из цикла.)

```
Insertion-Sort($A$)                                  стоимость   число раз
1  for $j \gets 2$ to $length[A]$                    $c_1$       $n$
2      do $key \gets A[j]$                           $c_2$       $n-1$
3         ▷ добавить $A[j]$ к отсортиро-
          ▷    ванной части $A[1..j-1]$.             0           $n-1$
4         $i \gets j-1$                              $c_4$       $n-1$
5         while $i > 0$ and $A[i] > key$             $c_5$       $\sum_{j=2}^{n} t_j$
6             do $A[i+1] \gets A[i]$                 $c_6$       $\sum_{j=2}^{n} (t_j - 1)$
7                $i \gets i-1$                       $c_7$       $\sum_{j=2}^{n} (t_j - 1)$
8         $A[i+1] \gets key$                         $c_8$       $n-1$
```

Строка стоимости c, повторённая m раз, даёт вклад cm в общее число операций. (Для количества использованной памяти этого сказать нельзя!) Сложив вклады всех строк, получим

$$\begin{aligned} T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j &+ \\ + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) &+ c_8(n-1). \end{aligned}$$

Как мы уже говорили, время работы процедуры зависит не только от n, но и от того, какой именно массив размера n подан ей на вход. Для процедуры Insertion-Sort наиболее благоприятен случай, когда массив уже отсортирован. Тогда цикл в строке 5 завершается после первой же проверки (поскольку $A[i] \leqslant key$ при $i = j-1$), так что все t_j равны 1, и общее время есть

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) = \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

Таким образом, в наиболее благоприятном случае время $T(n)$, необходимое для обработки массива размера n, является **линейной функцией** (linear function) от n, т.е. имеет вид $T(n) = an + b$ для некоторых констант a и b. (Эти константы определяются выбранными значениями $c_1, \ldots, c_8$.)

Если же массив расположен в обратном (убывающем) порядке, время работы процедуры будет максимальным: каждый элемент $A[j]$ придётся сравнить со всеми элементами $A[1] \ldots A[j-1]$. При этом $t_j = j$. Вспоминая, что

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1, \qquad \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

(см. гл. 3), получаем, что в худшем случае время работы процедуры равно

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) + \\ &\quad + c_6 \left(\frac{n(n-1)}{2} \right) + c_7 \left(\frac{n(n-1)}{2} \right) + c_8(n-1) = \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n - \\ &\quad - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

Теперь функция $T(n)$ — **квадратичная** (quadratic function), т.е. имеет вид $T(n) = an^2 + bn + c$. (Константы a, b и c снова определяются значениями c_1–c_8.)

Время работы в худшем случае и в среднем

Итак, мы видим, что время работы в худшем случае и в лучшем случае могут сильно различаться. Большей частью нас будет интересовать **время работы в худшем случае** (worst-case running time), которое определяется как максимальное время работы для входов данного размера. Почему? Вот несколько причин.

- Зная время работы в худшем случае, мы можем гарантировать, что выполнение алгоритма закончится за некоторое время, даже не зная, какой именно вход (данного размера) попадётся.
- На практике *"плохие"* входы (для которых время работы близко к максимуму) могут часто попадаться. Например, для базы данных плохим запросом может быть поиск отсутствующего элемента (довольно частая ситуация).
- Время работы в среднем может быть довольно близко к времени работы в худшем случае. Пусть, например, мы сортируем случайно расположенные n чисел в помощью процедуры Insertion-Sort. Сколько раз придётся выполнить цикл в строках 5–8? В среднем около половины элементов массива $A[1\,..\,j-1]$ больше $A[j]$, так что t_j в среднем можно считать равным $j/2$, и время $T(n)$ квадратично зависит от n.

В некоторых случаях нас будет интересовать также **среднее время работы** (average-case running time, expexted running time) алгоритма на входах данной длины. Конечно, эта величина зависит от выбранного распределения вероятностей (обычно рассматривается равномерное распределение), и на практике реальное распределение входов может оказаться совсем другим. (Иногда его можно преобразовать в равномерное, используя датчик случайных чисел.)

Порядок роста

Наш анализ времени работы процедуры Insertion-Sort был основан на нескольких упрощающих предположениях. Сначала мы предположили, что время выполнения i-й строки постоянно и равно c_i. Затем мы огрубили оценку до $an^2 + bn + c$. Сейчас мы пойдём ещё дальше и скажем, что время работы в худшем случае имеет **порядок роста** (rate of growth, order of growth) n^2, отбрасывая члены меньших порядков (линейные) и не интересуясь коэффициентом при n^2. Это записывают так: $T(n) = \Theta(n^2)$ (подробное объяснение обозначений мы отложим до следующей главы).

Алгоритм с меньшим порядком роста времени работы обычно предпочтителен: если, скажем, один алгоритм имеет время работы $\Theta(n^2)$, а другой — $\Theta(n^3)$, то первый более эффективен (по крайней мере для достаточно длинных входов; будут ли реальные входы таковыми — другой вопрос).

Упражнения

1.2-1 Будем сортировать массив из n элементов так: просмотрим его и найдём минимальный элемент, который скопируем в первую ячейку другого массива. Затем просмотрим его снова и найдём следующий элемент, и так далее. Такой способ сортировки можно назвать **сортировкой выбором** (selection sort). Запишите этот алгоритм с помощью псевдокода. Укажите время его работы в лучшем и худшем случаях, используя Θ-обозначения.

1.2-2 Вернёмся к алгоритму линейного поиска (упр. 1.1-3). Сколько сравнений потребуется в среднем этому алгоритму, если искомым элементом может быть любой элемент массива (с одинаковой вероятностью)? Каково время работы в худшем случае и в среднем? Как записать эти времена с помощью Θ-обозначений?

1.2-3 Дана последовательность чисел $x_1, x_2, \ldots, x_n$. Покажите, что за время $\Theta(n \log n)$ можно определить, есть ли в этой последовательности два одинаковых числа.

1.2-4 Даны коэффициенты $a_0, a_1, \ldots, a_{n-1}$ многочлена; требуется найти его значение в заданной точке x. Опишите естественный алгоритм, требующий времени $\Theta(n^2)$. Как выполнить вычисления за время $\Theta(n)$, не используя дополнительного массива? Используйте *"схему Горнера"*:

$$\sum_{i=0}^{n-1} a_i x^i = (\ldots(a_{n-1}x + a_{n-2})x + \ldots + a_1)x + a_0.$$

1.2-5 Как записать выражение $n^3/1000 - 100n^2 - 100n + 3$ с помощью Θ-обозначений?

1.2-6 Почти любой алгоритм можно немного изменить, радикально уменьшив время его работы в лучшем случае. Как?

1.3 Построение алгоритмов

Есть много стандартных приёмов, используемых при построении алгоритмов. Сортировка вставками является примером алгоритма, действующего **по шагам** (incremental approach): мы добавляем элементы один за другим к отсортированной части массива.

В этом разделе мы покажем в действии другой подход, который называют *"разделяй и властвуй"* (divide-and-conquer approach), и построим с его помощью значительно более быстрый алгоритм сортировки.

1.3.1 Принцип *"разделяй и властвуй"*

Многие алгоритмы по природе **рекурсивны** (recursive algorithms): решая некоторую задачу, они вызывают самих себя для решения её подзадач. Идея метода *"разделяй и властвуй"* состоит как раз в этом. Сначала задача разбивается на несколько подзадач меньшего размера. Затем эти задачи решаются (с помощью рекурсивного вызова — или непосредственно, если размер достаточно мал). Наконец, их решения комбинируются и получается решение исходной задачи.

Для задачи сортировки эти три этапа выглядят так. Сначала мы разбиваем массив на две половины меньшего размера. Затем мы сортируем каждую из половин отдельно. После этого нам остаётся соединить два упорядоченных массива половинного размера в один. Рекурсивное разбиение задачи на меньшие происходит до тех пор, пока размер массива не дойдёт до единицы (любой массив длины 1 можно считать упорядоченным).

Нетривиальной частью является соединение двух упорядоченных массивов в один. Оно выполняется с помощью вспомогательной процедуры $\textsc{Merge}(A,p,q,r)$. Параметрами этой процедуры являются массив A и числа p,q,r, указывающие границы сливаемых участков. Процедура предполагает, что $p \leqslant q < r$ и что участки $A[p\,..\,q]$ и $A[q+1\,..\,r]$ уже отсортированы, и **сливает** (merges) их в один участок $A[p\,..\,r]$.

Мы оставляем подробную разработку этой процедуры читателю (упр. 1.3-2), но довольно ясно, что время работы процедуры $\textsc{Merge}$ есть $\Theta(n)$, где n — общая длина сливаемых участков ($n = r-p+1$). Это легко объяснить на картах. Пусть мы имеем две стопки карт, и в каждой карты идут сверху вниз в возрастающем порядке. Как сделать из них одну? На каждом шаге мы берём меньшую из двух верхних карт и кладём её (рубашкой вверх) в результирующую стопку. Когда одна из исходных стопок становится пустой, мы добавляем все оставшиеся карты второй стопки к результирующей стопке. Ясно, что каждый шаг требует ограниченного числа действий, и общее число действий есть $\Theta(n)$.

Теперь напишем процедуру сортировки слиянием $\textsc{Merge-Sort}(A,p,r)$, которая сортирует участок $A[p\,..\,r]$ массива A, не меняя остальную часть массива. При $p \geqslant r$ участок содержит максимум один элемент, и тем самым уже отсортирован. В противном случае мы отыскиваем число q, которое делит участок на две примерно равные части $A[p\,..\,q]$ (содержит $\lceil n/2 \rceil$ элементов) и $A[q+1\,..\,r]$ (содержит $\lfloor n/2 \rfloor$ элементов). Здесь через $\lfloor x \rfloor$ мы обозначаем целую часть x (наибольшее целое число, меньшее или равное x), а через $\lceil x \rceil$ — наименьшее целое число, большее или равное x.

sorted sequence — отсортированная последовательность
merge — слияние
initial sequence — начальная последовательность

Рисунок 1.3 Сортировка слиянием для массива $A = \langle 5, 2, 4, 6, 1, 3, 2, 6 \rangle$.

```
Merge-Sort$(A, p, r)$
if $p < r$
   then $q \leftarrow \lfloor (p + r)/2 \rfloor$
        Merge-Sort$(A, p, q)$
        Merge-Sort$(A, q + 1, r)$
        Merge$(A, p, q, r)$
```

Весь массив теперь можно отсортировать с помощью вызова Merge-Sort$(A, 1, \mathit{length}[A])$. Если длина массива $n = \mathit{length}[A]$ есть степень двойки, то в процессе сортировки произойдёт слияние пар элементов в отсортированные участки длины 2, затем слияние пар таких участков в отсортированные участки длины 4 и так далее до n (на последнем шаге соединяются два отсортированных участка длины $n/2$). Этот процесс показан на рис. 1.3.

1.3.2 Анализ алгоритмов типа *"разделяй и властвуй"*

Как оценить время работы рекурсивного алгоритма? При подсчёте мы должны учесть время, затрачиваемое на рекурсивные вызовы, так что получается некоторое **рекуррентное соотношение** (recurrence equation). Далее следует оценить время работы, исходя из этого соотношения.

Вот примерно как это делается. Предположим, что алгоритм разбивает задачу размера n на a подзадач, каждая из которых имеет в b раз меньший размер. Будем считать, что разбиение требует времени $D(n)$, а соединение полученных решений — времени $C(n)$.

Тогда получаем соотношение для времени работы $T(n)$ на задачах размера n (в худшем случае): $T(n) = aT(n/b) + D(n) + C(n)$. Это соотношение выполнено для достаточно больших n, когда задачу имеет смысл разбивать на подзадачи. Для малых n, когда такое разбиение невозможно или не нужно, применяется какой-то прямой метод решения задачи. Поскольку n ограничено, время работы тоже не превосходит некоторой константы.

Анализ сортировки слиянием

Для простоты будем предполагать, что размер массива (n) есть степень двойки. (Как мы увидим в главе 4, это не очень существенно.) Тогда на каждом шаге сортируемый участок делится на две равные половины. Разбиение на части (вычисление границы) требует времени $\Theta(1)$, а слияние — времени $\Theta(n)$. Получаем соотношение

$$T(n) = \begin{cases} \Theta(1), & \text{если } n = 1, \\ 2T(n/2) + \Theta(n/2), & \text{если } n > 1. \end{cases}$$

Как мы увидим в главе 4, это соотношение влечёт $T(n) = \Theta(n \log n)$, где через $\log$ мы обозначаем двоичный логарифм (основание логарифмов, впрочем, не играет роли, так как приводит лишь к изменению константы). Поэтому для больших n сортировка слиянием эффективнее сортировки вставками, требующей времени $\Theta(n^2)$.

Упражнения

1.3-1 Следуя образцу рис. 1.3, показать работу сортировки слиянием для массива $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

1.3-2 Написать текст процедуры $\textsc{Merge}(A, p, q, r)$.

1.3-3 Докажите по индукции, что если

$$T(n) = \begin{cases} 2, & \text{если } n = 2, \\ 2T(n/2) + n, & \text{если } n = 2^k \text{ и } k > 1, \end{cases}$$

то $T(n) = n \log n$ (при всех n, являющихся степенями двойки).

1.3-4 Сортировку вставками можно оформить как рекурсивную процедуру: желая отсортировать $A[1\,..\,n]$, мы (рекурсивно) сортируем $A[1\,..\,n-1]$, а затем ставим $A[n]$ на правильное место в отсортированном массиве $A[1\,..\,n-1]$. Напишите рекуррентное соотношение для времени работы такой процедуры.

1.3-5 Возвращаясь к задаче поиска (упр. 1.1-3), заметим, что при поиске в отсортированном массиве мы можем сначала сравнить искомый элемент со средним элементом массива, узнать, в какой половине его следует искать, а затем применить ту же идею рекурсивно. Такой способ называется **двоичным поиском** (binary search). Напишите соответствующую программу, используя цикл или рекурсию. Объясните, почему время её работы есть $\Theta(\log n)$.

1.3-6 Заметим, что цикл **while** в строках 5–7 процедуры Insertion-Sort (разд. 1.1) просматривает элементы отсортированного участка $A[1\,..\,j-1]$ подряд. Вместо этого можно было бы использовать двоичный поиск (упр. 1.3-5), чтобы найти место вставки за время $\Theta(\log n)$. Удастся ли таким образом сделать общее время работы равным $\Theta(n \log n)$?

1.3-7* Дан массив S из n действительных чисел, а также число x. Как за время $\Theta(n \log n)$ определить, можно ли представить x в виде суммы двух элементов массива S?

Замечания

Как мог бы сказать Козьма Прутков, хороший алгоритм подобен острому ножу — тот и другой достигают цели легко и просто. Другое сравнение: человек, пользующийся плохим алгоритмом, подобен повару, отбивающему мясо отвёрткой: едва съедобный и малопривлекательный результат достигается ценой больших усилий.

Часто разница между плохим и хорошим алгоритмом более существенна, чем между быстрым и медленным компьютером. Пусть мы хотим отсортировать массив из миллиона чисел. Что быстрее — сортировать его вставками на суперкомпьютере (100 миллионов операций в секунду) или слиянием на домашнем компьютере (1 миллион операций)? Пусть к тому же сортировка вставками написана на ассемблере чрезвычайно экономно, и для сортировки n чисел нужно, скажем, лишь $2n^2$ операций. В то же время алгоритм слиянием написан без особой заботы об эффективности и число операций есть $50n \log n$. Для сортировки миллиона чисел получаем

$$\frac{2 \cdot (10^6)^2 \text{ операций}}{10^8 \text{ операций в секунду}} = 20\,000 \text{ секунд} \approx 5{,}56 \text{ часов}$$

для суперкомпьютера и всего

$$\frac{50 \cdot (10^6) \log(10^6) \text{ операций}}{10^6 \text{ операций в секунду}} \approx 1\,000 \text{ секунд} \approx 17 \text{ минут}$$

для домашнего компьютера.

Мы видим, что разработка эффективных алгоритмов — не менее важная компьютерная технология, чем разработка быстрой электроники. В этой области также происходит заметный прогресс.

Упражнения

1.3-1 Пусть сортировки вставками и слиянием исполняются на одной и той же машине и требуют $8n^2$ и $64n \log n$ соответственно. Для каких значений n сортировка вставками является более эффективной? Как можно улучшить алгоритм сортировки слиянием?

1.3-2 При каком наименьшем значении n алгоритм, делающий $100n^2$ операций, эффективнее алгоритма, делающего 2^n операций?

Задачи

1-1 Сравнение времени работы

Пусть имеется алгоритм, решающий задачу размера n за $f(n)$ микросекунд. Каков максимальный размер задачи, которую он сможет решить за время t? Найти его для функций и времён, перечисленных в таблице.

	1 сек	1 мин	1 час	1 день	1 месяц	1 год	1 век
$\log n$							
$\sqrt{n}$							
n							
$n \log n$							
n^2							
n^3							
2^n							
$n!$							

1-2 Сортировка вставками для коротких кусков

Асимптотически сортировка слиянием быстрее сортировки вставками, но для малых n соотношение обратное. Поэтому имеет смысл достаточно короткие куски не разбивать дальше, а применять к ним сортировку вставками. Вопрос в том, где следует провести границу.

а. Пусть массив длины n разбит на k частей размера n/k. Покажите, что можно отсортировать все части по отдельности (с помощью сортировки вставками) за время $\Theta(nk)$.

б. Покажите, что после этого можно слить все части в один упорядоченный массив за время $\Theta(n \log(n/k))$.

в. Тем самым общее время работы такого смешанного алгоритма есть $\Theta(nk + n \log(n/k))$. Какова максимальная скорость роста k как функции от n, при котором это время по-прежнему есть $\Theta(n \log n)$?

г. Как бы вы стали выбирать оптимальное значение k на практике?

1-3 Число инверсий

Пусть $A[1\,..\,n]$ — массив из n различных чисел. Нас будет интересовать количество **инверсий** (inversions) в этом массиве, т.е. число пар $i < j$, для которых $A[i] > A[j]$.

а. Укажите пять инверсий в массиве $\langle 2, 3, 8, 6, 1\rangle$.

б. Каково максимально возможное число инверсий в массиве длины n?

в. Как связано время работы алгоритма сортировки вставками и число инверсий? Объясните свой ответ.

г. Постройте алгоритм, который считает число инверсий в массиве длины n за время $\Theta(n \log n)$. (Указание: Модифицируйте алгоритм сортировки слиянием.)

Замечания

Есть множество хороших книг о построении алгоритмов. Вот некоторые из них: Ахо, Хопкрофт и Ульман [4,5], Баас [14], Брассар и Бретли [14], Хоровиц и Сахни [105], Кнут [121, 122, 123], Манбер [142], Мельхорн [144, 145, 146], Пурдом и Браун [164], Рейнгольд, Нивергельт и Део [167], Седжвик [175], Уилф [201]. Практические аспекты разработки эффективных алгоритмов: Бентли [24,25], Гоннет [90].

В 1968 году Кнут опубликовал первый из трёх томов серии *Искусство программирования для ЭВМ* [121, 122, 123], который стал началом новой эпохи в науке об эффективных алгоритмах. Все три тома до сих пор остаются незаменимым справочником. Как пишет Кнут, слово *"алгоритм"* происходит от имени арабского математика девятого века Ал-Хорезми (al-Khowârizmî, или al-Khwârizmî).

Ахо, Хопкрофт и Ульман [4] указали на важность асимптотического анализа времени работы как средства сравнения эффективности алгоритмов. Они широко использовали рекуррентные соотношения для получения оценок времени работы.

Книга Кнута [123] содержит исчерпывающее изложение множества алгоритмов сортировки. Он сравнивает различные алгоритмы, точно подсчитывая число различных шагов (мы делали это для сортировки сравнением). Рассматриваются различные варианты сортировки вставками, включая сортировку Шелла (D. L. Shell), которая использует сортировку подпоследовательностей с постоянным шагом для уменьшения числа операций.

Сортировка слиянием также описана в книге Кнута, который указывает, что механическое устройство для слияния двух стопок перфокарт за один проход было изобретено в 1938 году. По-видимому, Джон фон Нейман (J. von Neumann), один из основателей информатики, написал программу сортировки слиянием для компьютера EDVAC в 1945 году.

I Математические основы анализа алгоритмов

Введение

В этой части собраны сведения из математики, которые используются при анализе алгоритмов. Мы советуем бегло просмотреть её и перейти к следующим главам, возвращаясь к просмотренному по мере надобности.

В главе 2 мы вводим понятия и обозначения, связанные с асимптотикой функций (Θ и др.), а также некоторые другие. Наша цель здесь не рассказать о чём-то новом, а просто согласовать обозначения и терминологию.

В главе 3 приводятся различные методы вычисления и оценки сумм (подробное изложение можно найти в любом учебнике математического анализа).

Глава 4 посвящена преобразованию рекуррентных соотношений в явные оценки. Мы формулируем и доказываем общее утверждение такого рода (теорема 4.1), которого в большинстве случаев оказывается достаточно. Его доказательство довольно длинно, но в дальнейшем не используется, так что при первом чтении его можно пропустить.

В главе 5 мы даём определения различных понятий, связанных с множествами, отношениями, функциями, графами и деревьями, и вводим соответствующие обозначения.

Глава 6 посвящена основным понятиям комбинаторики и теории вероятностей. Большая часть книги не использует этого материала, так что при первом чтении эту главу (особенно последние разделы) можно смело пропустить, возвращаясь к пропущенному по мере надобности.

2 Скорость роста функций

Сравнивая два алгоритма сортировки в главе 1, мы установили, что время работы одного (сортировка слиянием) примерно пропорционально n^2, а другого (сортировка вставками) — $n \lg n$. Каковы бы ни были коэффициенты пропорциональности, для достаточно больших n первый алгоритм работает быстрее.

Анализируя алгоритм, можно стараться найти точное число выполняемых им действий. Но в большинстве случаев игра не стоит свеч, и достаточно оценить **асимптотику** роста времени работы алгоритма при стремлении размера входа к бесконечности (asymptotic efficiency). Если у одного алгоритма скорость роста меньше, чем у другого, то в большинстве случаев он будет эффективнее для всех входов, кроме совсем коротких. (Хотя бывают и исключения.)

2.1 Асимптотические обозначения

Хотя во многих случаях эти обозначения используются неформально, полезно начать с точных определений.

Θ-обозначение

В главе 2 мы говорили, что время $T(n)$ работы алгоритма сортировки вставками на входах длины n есть $\Theta(n^2)$. Точный смысл этого утверждения такой: найдутся такие константы $c_1, c_2 > 0$ и такое число n_0, что $c_1 n^2 \leqslant T(n) \leqslant c_2 n^2$ при всех $n \geqslant n_0$. Вообще, если $g(n)$ — некоторая функция, то запись $f(n) = \Theta(g(n))$ означает, что найдутся такие $c_1, c_2 > 0$ и такое n_0, что $0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)$ для всех $n \geqslant n_0$ (см. рис. 2.1). (Запись $f(n) = \Theta(g(n))$ читается так: *"эф от эн есть тэта от же от эн"*.)

[!!!!!!!!! Рисунок 2.1 - подпись к нему:]

Иллюстрации к определениям $f(n) = \Theta(g(n))$, $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$.

Разумеется, это обозначение следует употреблять с осторожностью: установив, что $f_1(n) = \Theta(g(n))$ и $f_2(n) = \Theta(g(n))$, не следует заключать, что $f_1(n) = f_2(n)$!

Определение $\Theta(g(n))$ предполагает, что функции $f(n)$ и $g(n)$ **асимптотически неотрицательны**, т. е. неотрицательны для достаточно больших значений n. Заметим, что если функции f и g строго положительны, то можно исключить n_0 из определения (изменив константы c_1 и c_2 так, чтобы для малых n неравенство также выполнялось).

Если $f(n) = \Theta(g(n))$, то говорят, что $g(n)$ является **асимптотически точной** оценкой для $f(n)$. На самом деле это отношение симметрично: если $f(n) = \Theta(g(n))$, то $g(n) = \Theta(f(n))$.

Вернёмся к примеру из главы 1 и проверим, что $(1/2)n^2 - 3n = \Theta(n^2)$. Согласно определению, надо указать положительные константы c_1, c_2 и число n_0 так, чтобы неравенства

$$c_1 n^2 \leqslant \frac{1}{2}n^2 - 3n \leqslant c_2 n^2$$

выполнялись для всех $n \geqslant n_0$. Разделим на n^2:

$$c_1 \leqslant \frac{1}{2} - \frac{3}{n} \leqslant c_2$$

Видно, что выполнения второго неравенства достаточно положить $c_2 = 1/2$. Первое будет выполнено, если (например) $n_0 = 7$ и $c_1 = 1/14$.

Другой пример использования формального определения: покажем, что $6n^3 \neq \Theta(n^2)$. В самом деле, пусть найдутся такие c_2 и n_0, что $6n^3 \leqslant c_2 n^2$ для всех $n \geqslant n_0$. Но тогда $n \leqslant c_2/6$ для всех $n \geqslant n_0$ — что явно не так.

Отыскивая асимптотически точную оценку для суммы, мы можем отбрасывать члены меньшего порядка, которые при больших n становятся малыми по сравнению с основным слагаемым. Заметим также, что коэффициент при старшем члене роли не играет (он может повлиять только на выбор констант c_1 и c_2). Например, рассмотрим квадратичную функцию $f(n) = an^2 + bn + c$, где a, b, c — некоторые константы и $a > 0$. Отбрасывая члены младших порядков и коэффициент при старшем члене, находим, что $f(n) = \Theta(n^2)$. Чтобы убедиться в этом формально, можно положить $c_1 = a/4$, $c_2 = 7a/4$ и $n_0 = 2 \cdot \max((|b|/a), \sqrt{|c|/a})$ (проверьте, что требования действительно выполнены). Вообще, для любого полинома $p(n)$ степени d с положительным старшим коэффициентом имеем $p(n) = \Theta(n^d)$ (задача 2-1).

Упомянем важный частный случай использования Θ-обозначений: $\Theta(1)$ обозначает ограниченную функцию, отделённую от нуля некоторый положительной константой при достаточно больших значениях аргумента. (Из контекста обычно ясно, что именно считается аргументом функции.)

O- и Ω-обозначения

Запись $f(n) = \Theta(g(n))$ включает в себя две оценки: верхнюю и нижнюю. Их можно разделить. Говорят, что $f(n) = O(g(n))$, если найдётся такая константа $c > 0$ и такое число n_0, что $0 \leqslant f(n) \leqslant cg(n)$ для всех $n \geqslant n_0$ Говорят, что $f(n) = \Omega(g(n))$, если найдется такая константа $c > 0$ и такое число n_0, что $0 \leqslant cg(n) \leqslant f(n)$ для всех $n \geqslant n_0$. Эти записи читаются так: *"эф от эн есть о большое от же от эн"*, *"эф от эн есть омéга большая от же от эн"*.

По-прежнему мы предполагаем, что функции f и g неотрицательны для достаточно больших значений аргумента. Легко видеть (упр. 2.1-5), что выполнены следующие свойства:

Теорема 2.1. *Для любых двух функций $f(n)$ и $g(n)$ свойство $f(n) = \Theta(g(n))$ выполнено тогда и только тогда, когда $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$.*

Для любых двух функций свойства $f(n) = O(g(n))$ и $g(n) = \Omega(f(n))$ равносильны.

Как мы видели, $an^2 + bn + c = \Theta(n^2)$ (при положительных a). Поэтому $an^2 + bn + c = O(n^2)$. Другой пример: при $a > 0$ можно написать $an + b = O(n^2)$ (положим $c = a + |b|$ и $n_0 = 1$). Заметим, что в этом случае $an + b \neq \Omega(n^2$ и $an + b \neq \Theta(n^2)$.

Асимптотические обозначения (Θ, O и Ω) часто употребляются внутри формул. Например, в главе 1 мы получили рекуррентное соотношение

$$T(n) = 2T(n/2) + \Theta(n)$$

для времени работы сортировки слиянием. Здесь $\Theta(n)$ обозначает некоторую функцию, про которую нам важно знать лишь, что она не меньше $c_1 n$ и не больше $c_2 n$ для некоторых положительных c_1 и c_2 и для всех достаточно больших n.

Часто асимптотические обозначения употребляются не вполне формально, хотя их подразумеваемый смысл обычно ясен из контекста. Например, мы можем написать выражение

$$\sum_{i=1}^{n} O(i)$$

имея в виду сумму $h(1) + h(2) + \ldots + h(n)$, где $h(i)$ — некоторая функция, для которой $h(i) = O(i)$. Легко видеть, что сама эта сумма как функция от n есть $O(n^2)$.

Типичный пример использования асимптотических обозначений — цепочка равенств наподобие $2n^2{+}3n{+}1 = 2n^2{+}\Theta(n) = \Theta(n^2)$. Второе из этих равенств ($2n^2 + \Theta(n) = \Theta(n^2)$) понимается при этом так: какова бы ни была функция $h(n) = \Theta(n)$ в левой части, сумма $2n^2 + h(n)$ есть $\Theta(n^2)$.

o- и ω-обозначения

Запись $f(n) = O(g(n))$ означает, что с ростом n отношение $f(n)/g(n)$ остаётся ограниченным. Если к тому же

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0, \tag{2.1}$$

то мы пишем $f(n) = o(g(n))$ (читается *"эф от эн есть о малое от же от эн"*). Формально говоря, $f(n) = o(g(n))$, если для всякого положительного $\varepsilon > 0$ найдётся такое n_0, что $0 \leqslant f(n) \leqslant \varepsilon g(n)$ при всех $n \geqslant n_0$. (Тем самым запись $f(n) = o(g(n))$ предполагает, что $f(n)$ и $g(n)$ неотрицательны для достаточно бо́льших n.)

Пример: $2n = o(n^2)$, но $2n^2 \neq o(n^2)$.

Аналогичным образом вводится ω-обозначение: говорят, что $f(n)$ есть $\omega(g(n))$ (*"эф от эн есть оме́га малая от же от эн"*), если для любого положительного c существует такое n_0, что $0 \leqslant cg(n) \leqslant f(n)$ при всех $n \geqslant n_0$. Другими словами, $f(n) = \omega(g(n))$ означает, что $g(n) = o(f(n))$.

Пример: $n^2/2 = \omega(n)$, но $n^2/2 \neq \omega(n^2)$.

Сравнение функций

Введённые нами определения обладают некоторыми свойствами транзитивности, рефлексивности и симметричности:

Транзитивность:

$f(n) = \Theta(g(n))$ и $g(n) = \Theta(h(n))$ влечёт $f(n) = \Theta(h(n))$,
$f(n) = O(g(n))$ и $g(n) = O(h(n))$ влечёт $f(n) = O(h(n))$,
$f(n) = \Omega(g(n))$ и $g(n) = \Omega(h(n))$ влечёт $f(n) = \Omega(h(n))$,
$f(n) = o(g(n))$ и $g(n) = o(h(n))$ влечёт $f(n) = o(h(n))$,
$f(n) = \omega(g(n))$ и $g(n) = \omega(h(n))$ влечёт $f(n) = \omega(h(n))$.

Рефлексивность:

$f(n) = \Theta(f(n)), \quad f(n) = O(f(n)), \quad f(n) = \Omega(f(n))$.

Симметричность:

$f(n) = \Theta(g(n))$ если и только если $g(n) = \Theta(f(n))$.

Обращение:

$f(n) = O(g(n))$ если и только если $g(n) = \Omega(f(n))$,
$f(n) = o(g(n))$ если и только если $g(n) = \omega(f(n))$.

Можно провести такую параллель: отношения между функциями f и g подобны отношениям между числами a и b:

$$\begin{array}{lcl} f(n) = O(g(n)) & \approx & a \leqslant b \\ f(n) = \Omega(g(n)) & \approx & a \geqslant b \\ f(n) = \Theta(g(n)) & \approx & a = b \\ f(n) = o(g(n)) & \approx & a < b \\ f(n) = \omega(g(n)) & \approx & a > b \end{array}$$

Параллель эта, впрочем, весьма условна: свойства числовых неравенств не переносятся на функции. Например, для любых двух чисел a и b всегда или $a \leqslant b$, или $a \geqslant b$, однако нельзя утверждать, что для любых двух (положительных) функций $f(n)$ и $g(n)$ или $f(n) = O(g(n))$, или $f(n) = \Omega(g(n))$. В самом деле, можно проверить, что ни одно из этих двух соотношений не выполнено для $f(n) = n$ и $g(n) = n^{1+\sin n}$ (показатель степени в выражении для $g(n)$ меняется в интервале от 0 до 2). Заметим ещё, что для чисел $a \leqslant b$ влечёт $a < b$ или $a = b$, в то время как для функций $f(n) = O(g(n))$ не влечёт $f(n) = o(g(n))$ или $f(n) = \Theta(g(n))$.

Упражнения

2.1-1 Пусть $f(n)$ и $g(n)$ неотрицательны для достаточно больших n. Покажите, что $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

2.1-2 Покажите, что

$$(n+a)^b = \Theta(n^b) \tag{2.2}$$

для любого вещественного a и для любого $b > 0$.

2.1-3 Почему утверждение "*время работы алгоритма A не меньше $O(n^2)$*" не имеет смысла?

2.1-4 Можно ли утверждать, что $2^{n+1} = O(2^n)$? Что $2^{2n} = O(2^n)$?

2.1-5 Докажите теорему 2.1.

2.1-6 Приведите пример функций $f(n)$ и $g(n)$, для которых $f(n) = O(g(n))$, но $f(n) \neq o(g(n))$ и $f(n) \neq \Theta(g(n))$.

2.1-7 Покажите, что свойства $f(n) = o(g(n))$ и $f(n) = \omega(g(n))$ не могут быть выполнены одновременно.

2.1-8 Асимптотические обозначения могут быть введены и для функций, зависящих от нескольких параметров. Говорят, что $f(m,n) = O(g(m,n))$, если найдутся n_0, m_0 и положительное c, для которых $0 \leqslant f(m,n) \leqslant cg(m,n)$ для всех $n \geqslant n_0$ и $m \geqslant m_0$. Дайте аналогичные определения для $\Theta(g(m,n))$ и $\Omega(g(m,n))$.

2.2 Стандартные функции и обозначения

Монотонность

Говорят, что функция $f(n)$ **монотонно возрастает** (is monotonically increasing), если $f(m) \leqslant f(n)$ при $m \leqslant n$. Говорят, что функция $f(n)$

монотонно убывает (is monotonically decreasing), если $f(m) \geqslant f(n)$ при $m \leqslant n$. Говорят, что функция $f(n)$ **строго возрастает** (is strictly increasing), если $f(m) < f(n)$ при $m < n$. Говорят, что функция $f(n)$ **строго убывает** (is strictly decreasing), если $f(m) > f(n)$ при $m < n$.

Целые приближения снизу и сверху

Для любого вещественного числа x через $\lfloor x \rfloor$ (the floor of x) мы обозначаем его целую часть, т.е. наибольшее целое число, не превосходящее x. Симметричным образом $\lceil x \rceil$ (the ceiling of x) обозначает наименьшее целое число, не меньшее x. Очевидно,

$$x - 1 < \lfloor x \rfloor \leqslant x \leqslant \lceil x \rceil < x + 1$$

для любого x. Кроме того,

$$\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$$

для любого целого n. Наконец, для любого x и для любых целых положительных a и b имеем

$$\lceil \lceil x/a \rceil / b \rceil = \lceil x/ab \rceil \tag{2.3}$$

и

$$\lfloor \lfloor x/a \rfloor / b \rfloor = \lfloor x/ab \rfloor \tag{2.4}$$

(чтобы убедиться в этом, полезно заметить, что для любого z и для целого n свойства $n \leqslant z$ и $n \leqslant \lfloor z \rfloor$ равносильны).

Функции $x \mapsto \lfloor x \rfloor$ и $x \mapsto \lceil x \rceil$ монотонно возрастают.

Многочлены

Многочленом (полиномом) степени d от переменной n (polynomial in n of degree d) называют функцию

$$p(n) = \sum_{i=0}^{d} a_i n_i$$

(d — неотрицательное целое число). Числа $a_0, a_1, \ldots, a_d$ называют **коэффициентами** (coefficients) многочлена. Мы считаем, что **старший коэффициент** a_d не равен нулю (если это не так, уменьшим d — это можно сделать, если только многочлен не равен нулю тождественно).

Для больших значений n знак многочлена $p(n)$ определяется старшим коэффициентом (остальные члены малы по сравнению с ним), так что при $a_d > 0$ многочлен $p(n)$ асимптотически положителен (положителен при больши́х n) и можно написать $p(n) = \Theta(n^d)$.

При $a \geqslant 0$ функция $n \mapsto n^a$ монотонно возрастает, при $a \leqslant 0$ — монотонно убывает. Говорят, что функция $f(n)$ **полиномиально ограничена**, если $f(n) = n^{O(1)}$, или, другими словами, если $f(n) = O(n^k)$ для некоторой константы k (см. упр. 2.2-2).

Экспоненты

Для любых вещественных m, n и $a \neq 0$ имеем

$$\begin{aligned} a^0 &= 1, & (a^m)^n &= a^{mn}, \\ a^1 &= a, & (a^m)^n &= (a^n)^m, \\ a^{-1} &= 1/a, & a^m a^n &= a^{m+n}. \end{aligned}$$

При $a \geqslant 1$ функция $n \mapsto a^n$ монотонно возрастает.

Мы будем иногда условно полагать $0^0 = 1$.

Функция $n \mapsto a^n$ называется показательной функцией, или экспонентой (exponential). При $a > 1$ показательная функция растёт быстрее любого полинома: каково бы ни было b,

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0 \tag{2.5}$$

или, другими словами, $n^b = o(a^n)$. Если в качестве основания степени взять число $e = 2{,}71828\ldots$, то экспоненту можно записать в виде ряда

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots = \sum_{k=0}^{\infty} \frac{x^k}{k!} \tag{2.6}$$

где $k! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot k$ (см. ниже о факториалах).

Для всех вещественных x выполнено неравенство

$$e^x \geqslant 1 + x \tag{2.7}$$

которое обращается в равенство лишь при $x = 0$. При $|x| \leqslant 1$ можно оценить e^x сверху и снизу так:

$$1 + x \leqslant e^x \leqslant 1 + x + x^2 \tag{2.8}$$

Можно сказать, что $e^x = 1 + x + \Theta(x^2)$ при $x \to 0$, имея в виду соответствующее истолкование обозначения Θ (в котором $n \to \infty$ заменено на $x \to 0$).

При всех x выполнено равенство $\lim_{n \to \infty} \left(1 + \frac{x}{n}\right)^n = e^x$.

Логарифмы

Мы будем использовать такие обозначения:

$$\begin{aligned} \lg n &= \log_2 n && \text{(двоичный логарифм)},\\ \ln n &= \log_e n && \text{(натуральный логарифм)},\\ \lg^k n &= (\lg n)^k, &&\\ \lg\lg n &= \lg(\lg n) && \text{(повторный логарифм)}. \end{aligned}$$

Мы будем считать, что в формулах знак логарифма относится лишь к непосредственно следующему за ним выражению, так что $\lg n + k$ есть $\lg(n) + k$ (а не $\lg(n+k)$). При $b > 1$ функция $n \mapsto \log_b n$ (определённая при положительных n) строго возрастает.

Следующие тождества верны при всех $a > 0$, $b > 0$, $c > 0$ и при всех n (если только основания логарифмов не равны 1):

$$\begin{aligned} a &= b^{\log_b a},\\ \log_c(ab) &= \log_c a + \log_c b,\\ \log_b a^n &= n\log_b a,\\ \log_b a &= \frac{\log_c a}{\log_c b} \end{aligned} \qquad \begin{aligned} \log_b(1/a) &= -\log_b a\\ \log_b a &= \frac{1}{\log_a b}\\ a^{\log_b c} &= c^{\log_b a} \end{aligned} \tag{2.9}$$

Изменение основания у логарифма умножает его на константу, поэтому в записи типа $O(\log n)$ можно не уточнять, каково основание логарифма. Мы будем чаще всего иметь дело с двоичными логарифмами (они появляются, когда задача делится на две части) и потому оставляем за ними обозначение lg.

Для натурального логарифма есть ряд (который сходится при $|x| < 1$):

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \ldots$$

При $x > -1$ справедливы неравенства

$$\frac{x}{1+x} \leqslant \ln(1+x) \leqslant x \tag{2.10}$$

которые обращаются в равенства лишь при $x = 0$.

Говорят, что функция $f(n)$ **ограничена полилогарифмом** (is polylogaritmically bounded), если $f(n) = \lg^{O(1)} n$. Предел (2.5) после подстановок $n = \lg m$ и $a = 2^c$ даёт

$$\lim_{m\to\infty} \frac{\lg^b m}{(2^c)^{\lg m}} = \lim_{m\to\infty} \frac{\lg^b m}{m^c} = 0$$

и, таким образом, $\lg^b n = o(n^c)$ для любой константы $c > 0$. Другими словами, любой полином растёт быстрее любого полилогарифма.

Факториалы

Запись $n!$ (читается *"эн факториал"*, "n factorial") обозначает произведение всех чисел от 1 до n. Полагают $0! = 1$, так что $n! = n \cdot (n-1)!$ при всех $n = 1, 2, 3, \ldots$.

Сразу же видно, что $n! \leqslant n^n$ (каждый из сомножителей не больше n). Более точная оценка даётся **формулой Стирлинга** (Stirling's approximation), которая гласит, что

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n (1 + \Theta(1/n)) \tag{2.11}$$

Из формулы Стирлинга следует, что

$$\begin{aligned} n! &= o(n^n), \\ n! &= \omega(2^n), \\ \lg(n!) &= \Theta(n \lg n). \end{aligned}$$

Справедлива также следующая оценка:

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \leqslant n! \leqslant \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{1/12n}. \tag{2.12}$$

Итерации логарифма

Мы используем обозначение $\log^* n$ (*"логарифм со звёздочкой от эн"*) для функции: называемой **итерированным логарифмом** (iterated logarithm). Эта функция определяется так. Вначале рассмотрим i-ую итерацию логарифма, функцию $\lg^{(i)}$, определённую так: $\lg^{(0)} n = n$ и $\lg^{(i)}(n) = \lg(\lg^{(i-1)} n)$ при $i > 0$. (Последнее выражение определено, если $\lg^{(i-1)} n$ определено и положительно.) Будьте внимательны: обозначения $\lg^i n$ и $\lg^{(i)} n$ внешне похожи, но означают совершенно разные функции.

Теперь $\lg^* n$ определяется как минимальное число $i \geqslant 0$, при котором $\lg^{(i)} n \leqslant 1$. Другими словами, $\lg^* n$ — это число раз, которое нужно применить функцию lg, чтобы из n получить число, не превосходящее 1.

Функция $\lg *n$ растёт исключительно медленно:

$$\begin{aligned} \lg^* 2 &= 1, \\ \lg^* 4 &= 2, \\ \lg^* 16 &= 3, \\ \lg^* 65536 &= 4, \\ \lg^* 2^{65536} &= 5. \end{aligned}$$

Поскольку число атомов в наблюдаемой части Вселенной оценивается как 10^{80}, что много меньше 2^{65536}, то значения n, для которых $\lg^* n > 5$, вряд ли могут встретиться.

Числа Фибоначчи

Последовательность **чисел Фибоначчи** (Fibonacci numbers) определяется рекуррентным соотношением:

$$F_0 = 0, \quad F_1 = 1, \quad F_i = F_{i-1} + F_{i-2} \text{ при } i \geqslant 2 \tag{2.13}$$

Другими словами, в последовательности Фибоначчи

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55\ \ldots$$

каждое число равно сумме двух предыдущих. Числа Фибоначчи связаны с так называемым отношением **золотого сечения** (golden ratio) φ и с сопряжённым с ним числом $\hat{\varphi}$:

$$\begin{aligned} \varphi &= \frac{1+\sqrt{5}}{2} = 1{,}61803\ldots, \\ \hat{\varphi} &= \frac{1-\sqrt{5}}{2} = -0{,}61803\ldots \end{aligned} \tag{2.14}$$

Именно, имеет место формула

$$F_i = \frac{\varphi^i - \hat{\varphi}^i}{\sqrt{5}} \tag{2.15}$$

которую можно доказать по индукции (упр. 2.2-7). Поскольку $|\hat{\varphi}| < 1$, слагаемое $|\hat{\varphi}^i/\sqrt{5}|$ меньше $1/\sqrt{5} < 1/2$, так что F_i равно числу $\varphi^i/\sqrt{5}$, округлённому до ближайшего целого.

Число F_i быстро (экспоненциально) растёт с ростом i.

Упражнения

2.2-1 Покажите, что для монотонно возрастающих функций $f(n)$ и $g(n)$ функции $f(n) + g(n)$ и $f(g(n))$ будут также монотонно возрастать. Если к тому же $f(n)$ и $g(n)$ неотрицательны при всех n, то и функция $f(n)g(n)$ будет монотонно возрастать.

2.2-2 Покажите, что $T(n) = n^{O(1)}$ тогда и только тогда, когда существует положительное k, при котором $T(n) = O(n^k)$ (считаем, что $T(n) > 1$).

2.2-3 Докажите равенства (2.9).

2.2-4 Докажите, что $\lg(n!) = \Theta(n \lg n)$ и что $n! = o(n^n)$.

2.2-5* Будет ли функция $\lceil \lg n \rceil!$ полиномиально ограниченной? Будет ли функция $\lceil \lg \lg n \rceil!$ полиномиально ограниченной?

2.2-6* Что больше (при больших n): $\lg(\lg^* n)$ или $\lg^*(\lg n)$?

2.2-7 Докажите по индукции формулу (2.15).

2.2-8 Докажите такую оценку для чисел Фибоначчи: $F_{i+2} \geqslant \varphi^i$ при $i \geqslant 0$ (здесь φ — отношение золотого сечения).

Задачи

2-1 Асимптотика многочленов

Пусть $p(n) = a_0 + a_1 n + \ldots + a_d n^d$ — многочлен степени d, причём $a_d > 0$. Докажите, что

а. $p(n) = O(n^k)$ при $k \geqslant d$.
б. $p(n) = \Omega(n^k)$ при $k \leqslant d$.
в. $p(n) = \Theta(n^k)$ при $k = d$.
г. $p(n) = o(n^k)$ при $k > d$.
д. $p(n) = \omega(n^k)$ при $k < d$.

2-2 Сравнение асимптотик

Для всех клеток следующей таблицы ответьте *"да"* или *"нет"* на вопрос о том, можно ли записать A как O, o, Ω, ω или Θ от B ($k \geqslant 1$, $\varepsilon > 0$, $c > 1$ — некоторые константы).

	A	B	O	o	Ω	ω	Θ
а.	$\lg^k n$	n^ε					
б.	n^k	c^n					
в.	$\sqrt{n}$	$n^{\sin n}$					
г.	2^n	$2^{n/2}$					
д.	$n^{\lg m}$	$m^{\lg n}$					
е.	$\lg(n!)$	$\lg(n^n)$					

2-3 Сравнение скорости роста

а. Расположите следующие 30 функций в порядке увеличения скорости роста (каждая функция есть O(следующая)) и отметьте, какие из этих функций на самом деле имеют одинаковую скорость роста (одна есть Θ от другой):

$$\begin{array}{cccccc}
\lg(\lg^* n) & 2^{\lg^* n} & (\sqrt{2})^{\lg n} & n^2 & n! & (\lg n)! \\
(3/2)^n & n^3 & \lg^2 n & \lg(n!) & 2^{2^n} & n^{1/\lg n} \\
\ln \ln n & \lg^* n & n \cdot 2^n & n^{\lg \lg n} & \ln n & 1 \\
2^{\lg n} & (\lg n)^{\lg n} & e^n & 4^{\lg n} & (n+1)! & \sqrt{\lg n} \\
\lg^* \lg n & 2^{\sqrt{2 \lg n}} & n & 2^n & n \lg n & 2^{2^{n+1}}
\end{array}$$

б. Укажите неотрицательную функцию $f(n)$, которая не сравнима ни с одной из функций g_i этой таблицы ($f(n)$ не есть $O(g_i(n))$ и $g_i(n)$ не есть $O(f(n))$).

2-4 Свойства асимптотических обозначений

Пусть функции $f(n)$ и $g(n)$ положительны при достаточно больших n. Можно ли утверждать, что

а. если $f(n) = O(g(n))$, то $g(n) = O(f(n))$?

б. $f(n) + g(n) = \Theta(\min(f(n), g(n)))$?

в. $f(n) = O(g(n))$ влечёт $\lg(f(n)) = O(\lg(g(n)))$, если $\lg(g(n)) > 0$ и $f(n) \geqslant 1$ для достаточно больших n?

г. $f(n) = O(g(n))$ влечёт $2^{f(n)} = O(2^{g(n)})$?

д. $f(n) = O((f(n))^2)$?

е. $f(n) = O(g(n))$ влечёт $g(n) = \Omega(f(n))$?

ж. $f(n) = \Theta(f(n/2))$?

з. $f(n) + o(f(n)) = \Theta(f(n))$?

2-5 Варианты асимптотических обозначений

В некоторых книгах Ω-обозначение используется в ином смысле. Мы будем использовать обозначение Ω^∞ для этого варианта, чтобы избежать путаницы. Пусть $f(n)$ и $g(n)$ — функции натурального аргумента. Говорят, что $f(n) = \Omega^\infty(g(n))$, если найдётся положительное число c, при котором $f(n) \geqslant cg(n) \geqslant 0$ для бесконечно многих натуральных n.

а. Покажите, что для любых двух функций $f(n)$ и $g(n)$, положительных при больших значениях n, выполнено либо $f(n) = O(g(n))$, либо $f(n) = \Omega^\infty(g(n))$, и что для нашего прежнего определения $\Omega(g(n))$ этого утверждать нельзя.

б. Каковы возможные достоинства и недостатки применения оценок вида Ω^∞ при исследовании времени работы алгоритмов?

Некоторые авторы в записи $f(n) = O(g(n))$ не требуют, чтобы функция $f(n)$ была асимптотически положительной. Будем использовать обозначение O' и скажем, что $f(n) = O'(g(n))$, если $|f(n)| = O(g(n))$.

в. Что происходит при таком определении с утверждениями теоремы 2.1?

Некоторые авторы используют ещё один вариант определения: будем говорить, что $f(n) = \tilde{O}(g(n))$, если найдутся положительные числа c и k, для которых $0 \leqslant f(n) \leqslant cg(n) \lg^k n$ для всех достаточно больших n.

г. Определите $\tilde{\Omega}$ и $\tilde{\Theta}$ аналогичным образом и докажите аналог теоремы 2.1.

2-6 Итерации

Мы итерировали логарифмическую функцию, но аналогичная операция возможна и для других функций. Пусть $f(n)$ — некоторая функция, причём $f(n) < n$. Определим $f^{(i)}(n)$, положив $f^{(0)}(n) = n$ и $f(i)(n) = f(f^{(i-1)}(n))$ при $i > 0$.

Для фиксированного числа c определим функцию $f_c^*(n)$ как минимальное $i \geqslant 0$, для которого $f^{(i)}(n) \leqslant c$. (Другими словами, f_c^* — это сколько раз нужно применять функцию f, чтобы из n получить число, не превосходящее c.) Заметим, что $f_c^*(n)$ определено далеко не всегда.

Для каждой из следующих функций $f(n)$ и значений c оцените f_c^* возможно точнее:

	$f(n)$	c	$f_c^*(n)$
а.	$\lg n$	1	
б.	$n-1$	0	
в.	$n/2$	1	
г.	$n/2$	2	
д.	$\sqrt{n}$	2	
е.	$\sqrt{n}$	1	
ж.	$n^{1/3}$	2	
з.	$n/\lg n$	2	

Замечания

Как считает Кнут [121], использование O-обозначений восходит к учебнику по теории чисел (P. Bachmann, 1892). Обозначение $o(g(n))$ было использовано Ландау (E. Landau) в 1909 году при обсуждении распределения простых чисел. Использование Ω- и Θ-обозначений рекомендовано Кнутом [124]: эти обозначения позволяют избежать технически некорректного употребления O-обозначений для нижних оценок (хотя многие люди продолжают так делать). Подробнее об асимптотических обозначениях см. Кнут [121, 124] и Брассар и Бретли [33].

Одни и те же обозначения порой используются в различных смыслах, хотя разница, как правило, оказывается несущественной. В частности, иногда сравниваемые функции не предполагаются неотрицательными при больших n (и сравниваются модули).

Существует много справочников, содержащих сведения об элементарных функциях: Абрамович и Стегун [1], Бейер [27]. Можно также взять любой учебник по анализу (см., например, Апостол [12] или Томас и Финни [192]). Книга Кнута [121] содержит много полезных математических сведений, используемых при анализе алгоритмов.

3 Суммирование

Если алгоритм содержит цикл (for, while), то время его работы является суммой времён отдельных шагов. Например, как мы знаем из раздела 1.2, выполнение j-го шага алгоритма сортировки вставками требует времени, пропорционального j. Поэтому общее время будет определяться суммой

$$\sum_{j=1}^{n} j,$$

которая есть $\Theta(n^2)$. Подобного рода суммы нам не раз встретятся (в частности, в главе 4 при анализе рекуррентных соотношений).

В разделе 3.1 перечислены основные свойства сумм. Некоторые из этих свойств доказываются в разделе 3.2; большинство пропущенных доказательств можно найти в учебниках по математическому анализу.

3.1 Суммы и их свойства

Для суммы $a_1 + a_2 + \ldots + a_n$ используют обозначение

$$\sum_{k=1}^{n} a_k.$$

При $n = 0$ значение суммы считается равным 0.

Как правило, нижний и верхний пределы суммирования — целые числа. (Если это не так, обычно подразумеваются целые части.)

В конечных суммах слагаемые можно произвольно переставлять.

В курсах анализа определяют сумму бесконечного **ряда** (series) $a_1 + a_2 + a_3 + \ldots$ или

$$\sum_{k=1}^{\infty} a_k$$

как предел последовательности частичных сумм

$$\lim_{n\to\infty}\sum_{k=1}^{n} a_k.$$

Если предела не существует, говорят, что ряд **расходится** (diverges); в противном случае он **сходится**. Если ряд $\sum_{k=1}^{\infty}|a_k|$ сходится, то ряд $\sum_{k=1}^{\infty} a_k$ называют **абсолютно сходящимся** (absolutely convergent series); всякий абсолютно сходящийся ряд сходится, но не наоборот. При перестановке членов абсолютно сходящегося ряда он остаётся абсолютно сходящимся и его сумма не меняется.

Линейность

Свойство линейности гласит, что

$$\sum_{k=1}^{n}(ca_k+b_k) = c\sum_{k=1}^{n} a_k + \sum_{k=1}^{n} b_k$$

для любого числа c и для любых конечных последовательностей $a_1,\ldots,a_n$ и $b_1,\ldots,b_n$. Почленно складывать и умножать на числа можно не только конечные суммы, но и сходящиеся бесконечные ряды.

Арифметические прогрессии

Сумма

$$\sum_{k=1}^{n} k = 1+2+\ldots+n,$$

которая возникла при анализе алгоритма сортировки вставками, является **арифметической прогрессией** (arithmetic series). Её значение равно

$$\sum_{k=1}^{n} k = \frac{1}{2}n(n+1) = \tag{3.1}$$

$$= \Theta(n^2). \tag{3.2}$$

Геометрические прогрессии

При $x \neq 1$ сумму **геометрической прогрессии** (geometric или exponential series)

$$\sum_{k=0}^{n} x^k = 1+x+x^2+\ldots+x^n$$

можно найти по формуле

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}. \tag{3.3}$$

Сумма бесконечно убывающей геометрической прогрессии (при $|x| < 1$) даётся формулой

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}. \tag{3.4}$$

Гармонический ряд

имеет вид $1 + 1/2 + 1/3 + \ldots + 1/n + \ldots$; его n-ая частичная сумма (nth harmonic number) равна

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k} = \ln n + O(1). \tag{3.5}$$

Почленное интегрирование и дифференцирование

Дифференцируя или интегрируя обе части известного тождества, можно получить новое. Например, дифференцируя тождество (3.4) и умножая результат на x, получаем

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}. \tag{3.6}$$

Суммы разностей

Для любой последовательности $a_0, a_1, \ldots, a_n$ можно записать тождество

$$\sum_{k=1}^{n} (a_k - a_{k-1}) = a_n - a_0 \tag{3.7}$$

(все промежуточные члены сокращаются). Такие суммы по-английски называют telescoping series. Аналогично,

$$\sum_{k=0}^{n-1} (a_k - a_{k+1}) = a_0 - a_n.$$

Этот приём позволяет просуммировать ряд $\sum_{k=1}^{n-1} \frac{1}{k(k+1)}$. Поскольку $\frac{1}{k(k+1)} = \frac{1}{k} - \frac{1}{k+1}$, получаем, что

$$\sum_{k=1}^{n-1} \frac{1}{k(k+1)} = \sum_{k=1}^{n-1} \left(\frac{1}{k} - \frac{1}{k+1}\right) = 1 - \frac{1}{n}.$$

Произведения

Произведение чисел $a_1, \ldots, a_n$ записывают как

$$\prod_{k=1}^{n} a_k .$$

При $n = 0$ значение defnпроизведения (product) считается равным 1. Логарифмирование превращает произведение в сумму:

$$\lg \prod_{k=1}^{n} a_k = \sum_{k=1}^{n} \lg a_k .$$

Упражнения

3.1-1 Вычислите $\sum_{k=1}^{n}(2k-1)$.

3.1-2* Покажите, используя формулу для частичных сумм гармонического ряда, что $\sum_{k=1}^{n} 1/(2k-1) = \ln(\sqrt{n}) + O(1)$.

3.1-3* Покажите, что $\sum_{k=0}^{\infty}(k-1)/2^k = 0$.

3.1-4* Вычислите сумму $\sum_{k=1}^{\infty}(2k+1)x^{2k}$.

3.1-5 Используя линейность суммы, сформулируйте и докажите утверждение о возможности перестановки асимптотического O-обозначения и суммирования.

3.1-6 Используя линейность суммы, сформулируйте и докажите утверждение о возможности перестановки асимптотического Ω-обозначения и суммирования.

3.1-7 Вычислите произведение $\prod_{k=1}^{n} 2 \cdot 4^k$.

3.1-8* Вычислите произведение $\prod_{k=2}^{n}(1 - 1/k^2)$.

3.2 Оценки сумм

Рассмотрим несколько приёмов, которые позволяют найти значение суммы (или хотя бы оценить эту сумму сверху или снизу).

Индукция

Если удалось угадать формулу для суммы, её легко проверить с помощью математической индукции. Пример: докажем, что сумма арифметической прогрессии $S_n = \sum_{k=1}^{n} k$ равна $n(n+1)/2$. При $n = 1$ это верно. Теперь предположим, что равенство $S_n = n(n+1)/2$ верно при некотором n и проверим его для $n+1$. В самом деле,

$$\sum_{k=1}^{n+1} k = \sum_{k=1}^{n} k + (n+1) = n(n+1)/2 + (n+1) = (n+1)(n+2)/2.$$

Индукцию можно использовать и для неравенств. Например, покажем, что сумма геометрической прогрессии $\sum_{k=0}^{n} 3^k$ есть $O(3^n)$. Точнее, мы покажем, что $\sum_{k=0}^{n} 3^k \leqslant c\cdot 3^n$ для некоторой константы c (которую мы выберем позднее). При $n = 0$ имеем $\sum_{k=0}^{0} 3^k = 1 \leqslant c\cdot 1$; это верно при $c \geqslant 1$. Предполагая справедливость оценки при некотором n, докажем её для $n+1$. Имеем:

$$\sum_{k=0}^{n+1} 3^k = \sum_{k=0}^{n} 3^k + 3^{n+1} \leqslant c3^n + 3^{n+1} = \left(\frac{1}{3} + \frac{1}{c}\right) c3^{n+1} \leqslant c3^{n+1}.$$

Последний переход законен, если $(1/3 + 1/c) \leqslant 1$, т. е. если $c \geqslant 3/2$. Поэтому $\sum_{k=0}^{n} 3^k = O(3^n)$, что и требовалось доказать.

Индукцией следует пользоваться аккуратно, особенно при доказательстве асимптотических оценок, поскольку тут легко ошибиться. Для примера "*докажем*", что $\sum_{k=1}^{n} k = O(n)$. Очевидно, $\sum_{k=1}^{1} k = O(1)$. Предполагая справедливость оценки при некотором n, докажем её для следующего значения n. В самом деле,

$$\sum_{k=1}^{n+1} k = \sum_{k=1}^{n} k + (n+1) = O(n) + (n+1) \overset{\text{[неверно!]}}{=} O(n+1).$$

Ошибка в том, что константа, подразумеваемая в обозначении $O(n)$, растёт вместе с n.

Почленные сравнения

Иногда можно получить верхнюю оценку для суммы, заменив каждый её член на больший (например, на наибольший из членов суммы). Так, простейшей оценкой сверху для суммы арифметической прогрессии $\sum_{k=1}^{n} k$ будет

$$\sum_{k=1}^{n} k \leqslant \sum_{k=1}^{n} n = n^2.$$

В общем случае

$$\sum_{k=1}^{n} a_k \leqslant n a_{\max},$$

где $a_{\max}$ — наибольшее из $a_1, \ldots, a_n$.

В некоторых случаях можно применить более точный метод — сравнение с геометрической прогрессией. Допустим, дан ряд $\sum_{k=0}^{n} a_k$ с положительными членами, для которого $a_{k+1}/a_k \leqslant r$ при всех $k \geqslant 0$ и некотором $r < 1$. Тогда $a_k \leqslant a_0 r^k$, и сумма может быть ограничена сверху бесконечной убывающей геометрической прогрессией:

$$\sum_{k=0}^{n} a_k \leqslant \sum_{k=0}^{\infty} a_0 r^k = a_0 \sum_{k=0}^{\infty} r^k = a_0 \frac{1}{1-r}.$$

Применим этот метод для суммы $\sum_{k=1}^{\infty} k3^{-k}$. Первый член равен $1/3$, отношение соседних членов равно

$$\frac{(k+1)3^{-(k+1)}}{k3^{-k}} = \frac{1}{3} \cdot \frac{k+1}{k} \leqslant \frac{2}{3}$$

для всех $k \geqslant 1$. Следовательно, каждый член суммы оценивается сверху величиной $(1/3)(2/3)^k$, так что

$$\sum_{k=1}^{\infty} k3^{-k} \leqslant \sum_{k=1}^{\infty} \frac{1}{3} \cdot \left(\frac{2}{3}\right)^k = \frac{1}{3} \cdot \frac{1}{1-\frac{2}{3}} = 1.$$

Важно, что отношение соседних членов не просто меньше 1, а ограничено некоторой константой $r < 1$, общей для всех членов ряда. Например, для гармонического ряда отношение $(k+1)$-го и k-го членов равно $\frac{k}{k+1} < 1$. Тем не менее

$$\sum_{k=1}^{\infty} \frac{1}{k} = \lim_{n\to\infty} \sum_{k=1}^{n} \frac{1}{k} = \lim_{n\to\infty} \Theta(\lg n) = \infty.$$

Здесь не существует константы r, отделяющей отношения соседних членов от единицы.

Разбиение на части

Можно разбить сумму на части и оценивать каждую часть по отдельности. В качестве примера получим нижнюю оценку для суммы арифметической прогрессии $\sum_{k=1}^{n} k$. Если мы оценим снизу каждый член единицей, то получим оценку $\sum_{k=1}^{n} k \geqslant n$. Разница с верхней оценкой $O(n^2)$ слишком велика, поэтому поступим по-другому:

$$\sum_{k=1}^{n} k = \sum_{k=1}^{n/2} k + \sum_{k=n/2+1}^{n} k \geqslant \sum_{k=1}^{n/2} 0 + \sum_{k=n/2+1}^{n} \frac{n}{2} = \left(\frac{n}{2}\right)^2.$$

Эта оценка уже асимптотически точна (отличается от верхней оценки, и тем самым от истинного значения, не более чем в константу раз).

Иногда полезно отбросить несколько первых членов последовательности, записав (для фиксированного k_0)

$$\sum_{k=0}^{n} a_k = \sum_{k=0}^{k_0} a_k + \sum_{k=k_0+1}^{n} a_k = \Theta(1) + \sum_{k=k_0+1}^{n} a_k.$$

Например, для ряда

$$\sum_{k=0}^{\infty} \frac{k^2}{2^k}$$

отношение последовательных членов

$$\frac{(k+1)^2/2^{k+1}}{k^2/2^k} = \frac{(k+1)^2}{2k^2}$$

не всегда меньше 1. Поэтому, чтобы применить метод сравнения с геометрической прогрессией, выделим три первых члена ряда. Для всех следующих членов ($k \geqslant 3$) отношение не превосходит 8/9, поэтому

$$\sum_{k=0}^{\infty} \frac{k^2}{2^k} = \sum_{k=0}^{2} \frac{k^2}{2^k} + \sum_{k=3}^{\infty} \frac{k^2}{2^k} \leqslant O(1) + \frac{9}{8} \sum_{k=0}^{\infty} \left(\frac{8}{9}\right)^k = O(1),$$

поскольку вторая сумма — бесконечно убывающая геометрическая прогрессия.

Разбиение ряда на части бывает полезно и в более сложных ситуациях. Оценим, например, частичные суммы гармонического ряда $H_n = \sum_{k=1}^{n} \frac{1}{k}$. Для этого разобьем отрезок от 1 до n на $\lfloor \lg n \rfloor$ частей $1 + (1/2 + 1/3) + (1/4 + 1/5 + 1/6 + 1/7) + \ldots$. Каждая часть не превосходит 1 (заменим все слагаемые в ней на первое из них), всего частей $\lfloor \lg n \rfloor + 1$ (последняя может быть неполной). Получаем, что

$$1 + 1/2 + 1/3 + \ldots + 1/n \leqslant \lg n + 1. \tag{3.8}$$

Сравнение с интегралами

Для монотонно возрастающей функции f мы можем заключить сумму $\sum_{k=m}^{n} f(k)$ между двумя интегралами:

$$\int_{m-1}^{n} f(x)\,dx \leqslant \sum_{k=m}^{n} f(k) \leqslant \int_{m}^{n+1} f(x)\,dx. \tag{3.9}$$

Рисунок 3.1 Сравнение суммы $\sum_{k=m}^{n} f(k)$ с интегралами. Внутри каждого прямоугольника записана его площадь. Общая площадь всех прямоугольников равна значению суммы. Значение интеграла равно площади закрашенной фигуры под кривой. (а) Сравнивая площади, получаем $\int_{m-1}^{n} f(x)\,dx \leqslant \sum_{k=m}^{n} f(k)$. (б) Сдвигая прямоугольники на единицу вправо, видим, что $\sum_{k=m}^{n} f(k) \leqslant \int_{m}^{n+1} f(x)\,dx$.

В самом деле, как видно из рис. 3.1, сумма (площадь всех прямоугольников) содержит одну закрашенную область (верхний рисунок) и содержится в другой (нижний рисунок). Аналогичное неравенство можно написать для монотонно убывающей функции f:

$$\int\limits_{m}^{n+1} f(x)\,dx \leqslant \sum_{k=m}^{n} f(k) \leqslant \int\limits_{m-1}^{n} f(x)\,dx. \tag{3.10}$$

Этим методом можно получить хорошие оценки для частичных сумм гармонического ряда: нижнюю оценку

$$\sum_{k=1}^{n} \frac{1}{k} \geqslant \int\limits_{1}^{n+1} \frac{dx}{x} = \ln(n+1) \tag{3.11}$$

и верхнюю оценку (для ряда без первого члена)

$$\sum_{k=2}^{n} \frac{1}{k} \leqslant \int\limits_{1}^{n} \frac{dx}{x} = \ln n,$$

откуда

$$\sum_{k=1}^{n} \frac{1}{k} \leqslant \ln n + 1. \tag{3.12}$$

Упражнения

3.2-1 Покажите, что сумма $\sum_{k=1}^{n} \frac{1}{k^2}$ ограничена сверху константой (не зависящей от n).

3.2-2 Найдите асимптотическую верхнюю оценку для суммы

$$\sum_{k=0}^{\lfloor \lg n \rfloor} \lceil n/2^k \rceil.$$

3.2-3 Разбивая сумму на части, покажите, что n-я частичная сумма гармонического ряда есть $\Omega(\lg n)$.

3.2-4 Заключите сумму $\sum_{k=1}^{n} k^3$ между двумя интегралами.

3.2-5 Почему при получении верхней оценки для частичной суммы гармонического ряда мы отбросили первый член и только потом применили оценку (3.10)?

Задачи

3-1 Оценки для сумм

Найдите асимптотически точные оценки для следующих сумм (считая $r \geqslant 0$ и $s \geqslant 0$ константами):

а. $\sum_{k=1}^{n} k^r$.

б. $\sum_{k=1}^{n} \lg^s k$.

в. $\sum_{k=1}^{n} k^r \lg^s k$.

Замечания

Книга Кнута [121] — прекрасный справочник по материалу этой главы. Основные свойства рядов можно найти в любом учебнике по математическому анализу (авторы отсылают англоязычного читателя к книгам [12] и [192]).

4 Рекуррентные соотношения

Оценивая время работы рекурсивной процедуры, мы часто приходим к соотношению, которое оценивает это время через время работы той же процедуры на входных данных меньшего размера. Такого рода соотношения называются **рекуррентными** (recurrences). Например, как мы видели в главе 1, время работы процедуры Merge-Sort описывается соотношением

$$T(n) = \begin{cases} \Theta(1), & \text{если } n = 1, \\ 2T(n/2) + \Theta(n), & \text{если } n > 1. \end{cases} \tag{4.1}$$

из которого вытекает (как мы увидим), что $T(n) = \Theta(n \lg n)$.

В этой главе предлагаются три способа, позволяющие решить рекуррентное соотношение, т. е. найти асимптотическую ("Θ" или "O") оценку для его решений. Во-первых, можно угадать оценку, а затем доказать её по индукции, подставляя угаданную формулу в правую часть соотношения (**метод подстановки**, substitution method). Во-вторых, можно *"развернуть"* рекуррентную формулу, получив при этом сумму, которую можно затем оценивать (**метод итераций**, iteration method). Наконец, мы приводим общий результат о рекуррентных соотношениях вида

$$T(n) = aT(n/b) + f(n),$$

где $a \geqslant 1$, $b > 1$ — некоторые константы, а $f(n)$ — заданная функция. Соответствующий результат мы будет называть **основной теоремой о рекуррентных соотношениях** (master theorem). Формулировка этой теоремы довольно длинна, зато во многих случаях она сразу приводит к ответу.

Технические детали

Формулируя и доказывая утверждения про рекуррентные соотношения, мы будем опускать некоторые технические подробности. Например, в приведённой выше формуле для процедуры Merge-Sort должны стоять целые части ($T(n)$ определено лишь при целых n):

$$T(n) = \begin{cases} \Theta(1), & \text{если } n = 1, \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n), & \text{если } n > 1. \end{cases} \qquad (4.2)$$

Кроме того, обычно мы будем опускать начальные условия, имея в виду, что $T(n) = \Theta(1)$ для небольших n. Таким образом, соотношение (4.1) запишется в виде

$$T(n) = 2T(n/2) + \Theta(n). \qquad (4.3)$$

Это позволительно, так как начальное условие (значение $T(1)$) влияет только на постоянный множитель, но не на порядок роста функции $T(n)$.

Такое небрежное обращение с деталями, конечно, требует осторожности, чтобы не упустить (пусть редких) случаев, в которых подобные детали влияют на результат. В этой главе мы разберём несколько примеров, показывающих, как восполнить пропущенные детали (см. доказательство теоремы 4.1 и задачу 4.5).

4.1 Метод подстановки

Идея проста: отгадать ответ и доказать его по индукции. Часто ответ содержит коэффициенты, которые надо выбрать так, чтобы рассуждение по индукции проходило.

Индуктивный метод применим и к нижним, и к верхним оценкам. В качестве примера найдём верхнюю оценку для функции, заданной соотношением

$$T(n) = 2T(\lfloor n/2 \rfloor) + n, \qquad (4.4)$$

которое аналогично (4.2) и (4.3). Можно предположить, что $T(n) = O(n \lg n)$, т.е. что $T(n) \leqslant cn \lg n$ для подходящего $c > 0$. Будем доказывать это по индукции. Пусть эта оценка верна для $\lfloor n/2 \rfloor$, т.е. $T(\lfloor n/2 \rfloor) \leqslant c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. Подставив её в соотношение, получим

$$\begin{aligned} T(n) &\leqslant 2(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \leqslant \\ &\leqslant cn \lg(n/2) + n = cn \lg n - cn \lg 2 + n = cn \lg n - cn + n \leqslant \\ &\leqslant cn \lg n. \end{aligned}$$

Последний переход законен при $c \geqslant 1$.

Для завершения рассуждения остаётся проверить базис индукции, т.е. доказать оценку для начального значения n. Тут мы сталкиваемся с тем, что при $n = 1$ правая часть неравенства обращается в нуль, каким бы ни взять c (поскольку $\lg 1 = 0$). Приходится вспомнить, что асимптотическую оценку достаточно доказать для всех n, начиная с некоторого. Подберём c так, чтобы оценка $T(n) \leqslant cn \lg n$ была верна при $n = 2$ и $n = 3$. Тогда для больших n можно рассуждать по индукции, и опасный случай $n = 1$ нам не встретится (поскольку $\lfloor n/2 \rfloor \geqslant 2$ при $n > 3$).

Как отгадать оценку?

Для этого нужен навык и немного везения. Вот несколько наводящих соображений.

Аналогия. Рассмотрим для примера соотношение

$$T(n) = 2T(\lfloor n/2 \rfloor + 17) + n.$$

которое отличается от (4.4) добавочным слагаемым 17 в правой части. Можно ожидать, однако, что эта добавка не может существенно изменить характер решения: при больших n разница между $\lfloor n/2 \rfloor + 17$ и $\lfloor n/2 \rfloor$ вряд ли так уж существенна. Можно предположить, что оценка $T(n) = O(n \lg n)$ остаётся в силе, а затем и доказать это по индукции (см. упр. 4.1-5).

Последовательные приближения. Можно начать с простых и грубых оценок, а затем уточнять их. Например, для соотношения (4.1) есть очевидная нижняя оценка $T(n) = \Omega(n)$ (поскольку справа есть член n), и верхняя оценка $T(n) = O(n^2)$ (которую легко доказать по индукции). Далее можно постепенно сближать их, стремясь получить асимптотически точные нижнюю и верхнюю оценки, отличающиеся не более чем в константу раз.

Тонкости

Иногда рассуждение по индукции сталкивается с трудностями, хотя ответ угадан правильно. Обычно это происходит потому, что доказываемое по индукции утверждение недостаточно сильно. В этом случае может помочь вычитание члена меньшего порядка.

Рассмотрим соотношение

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1.$$

Можно надеяться, что в этом случае $T(n) = O(n)$. Это действительно так. Попробуем, однако, доказать, что $T(n) \leqslant cn$ при подходящем выборе константы c. Индуктивное предположение даёт

$$T(n) \leqslant c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 = cn + 1,$$

а отсюда ни для какого c не вытекает неравенства $T(n) \leqslant cn$. В такой ситуации можно попытаться доказать более слабую оценку, например $O(n^2)$, но на самом деле в этом нет необходимости: наша догадка верна, надо только не ослабить, а усилить предположение индукции.

Новая гипотеза выглядит так: $T(n) \leqslant cn - b$ для некоторых констант b и c. Подстановка в правую часть даёт

$$T(n) \leqslant (c\lfloor n/2 \rfloor - b) + (c\lceil n/2 \rceil - b) + 1 = cn - 2b + 1 \leqslant cn - b$$

Последний переход законен при $b \geqslant 1$. Остаётся лишь выбрать константу c с учётом начальных условий.

Это рассуждение вызывает недоумение: если доказательство не проходит, не следует ли ослабить (увеличить) оценку, а не усиливать её? Но на самом деле ничего странного здесь нет — усилив оценку, мы получаем возможность воспользоваться более сильным индуктивным предположением.

Как делать не надо

Асимптотическая запись опасна при неумелом применении: вот пример неправильного "*доказательства*" оценки $T(n) = O(n)$ для соотношения (4.4). Предположим, что $T(n) \leqslant cn$, тогда можно записать

$$T(n) \leqslant 2(c\lfloor n/2 \rfloor) + n \leqslant cn + n = O(n).$$

Это рассуждение, однако, ничего не доказывает, так как индуктивный переход требует, чтобы в правой части было cn с той же самой константой c, а не абстрактное $O(n)$.

Замена переменных

Часто несложная замена переменных позволяет преобразовать рекуррентное соотношение к привычному виду. Например, соотношение

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \ln n$$

кажется довольно сложным, однако заменой переменных его легко упростить. Сделав замену $m = \lg n$, получим

$$T(2^m) = 2T(2^{m/2}) + m.$$

Обозначив $T(2^m)$ через $S(m)$, приходим к соотношению

$$S(m) = 2S(m/2) + m,$$

которое уже встречалось (4.4). Его решение: $S(m) = O(m \lg m)$. Возвращаясь к $T(n)$ вместо $S(m)$, получим

$$T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n).$$

Приведённое рассуждение требует, конечно, уточнений, поскольку пока что мы доказали оценку на $T(n)$ лишь для n, являющихся степенями двойки. Чтобы выйти из положения, можно определить $S(m)$ как максимум $T(n)$ по всем n, не превосходящим 2^m.

Упражнения

4.1-1 Покажите, что из $T(n) = T(\lceil n/2 \rceil) + 1$ следует, что $T(n) = O(\lg n)$.

4.1-2 Покажите, что из $T(n) = 2T(\lfloor n/2 \rfloor) + n$ вытекает $T(n) = \Omega(n \lg n)$, и тем самым $T(n) = \Theta(n \lg n)$.

4.1-3 Как обойти трудность с начальным значением $n = 1$ при исследовании соотношения (4.4), изменив доказываемое по индукции утверждение, но не меняя начального значения?

4.1-4 Покажите, что соотношение (4.2) для сортировки слиянием имеет решением $\Theta(n \lg n)$.

4.1-5 Покажите, что $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$ влечёт $T(n) = O(n \lg n)$.

4.1-6 Решите с помощью замены переменных соотношение $T(n) = 2T(\sqrt{n}) + 1$, (опуская подробности, связанные с тем, что значения переменных — не целые).

4.2 Преобразование в сумму

Как быть, если не удается угадать решение? Тогда можно, итерируя это соотношение (подставляя его само в себя), получить ряд, который можно оценивать тем или иным способом.

Для примера рассмотрим соотношение

$$T(n) = 3T(\lfloor n/4 \rfloor) + n.$$

Подставляя его в себя, получим:

$$\begin{aligned} T(n) &= n + 3T(\lfloor n/4 \rfloor) = n + 3(\lfloor n/4 \rfloor + 3T(\lfloor n/16 \rfloor)) \\ &= n + 3(\lfloor n/4 \rfloor + 3(\lfloor n/16 \rfloor + 3T(\lfloor n/64 \rfloor))) \\ &= n + 3\lfloor n/4 \rfloor + 9\lfloor n/16 \rfloor + 27T(\lfloor n/64 \rfloor), \end{aligned}$$

Мы воспользовались тем, что, согласно (2.4), $\lfloor \lfloor n/4 \rfloor /4 \rfloor = \lfloor n/16 \rfloor$ и $\lfloor \lfloor n/16 \rfloor /4 \rfloor = \lfloor n/64 \rfloor$. Сколько шагов надо сделать, чтобы дойти до начального условия? Поскольку после i-ой итерации справа окажется $T(\lfloor n/4^i \rfloor)$, мы дойдём до $T(1)$, когда $\lfloor n/4^i \rfloor = 1$, т.е. когда $i \geqslant \log_4 n$. Заметив, что $\lfloor n/4^i \rfloor \leqslant n/4^i$, мы можем оценить наш ряд убывающей геометрической прогрессией (плюс последний член, соответствующий $3^{\log_4 n}$ задачам ограниченного размера):

$$\begin{aligned} T(n) &\leqslant n + 3n/4 + 9n/16 + 27n/64 + \ldots + 3^{\log_4 n}\Theta(1) \\ &\leqslant n\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i + \Theta(n^{\log_4 3}) \\ &= 4n + o(n) = O(n) \end{aligned}$$

(Мы заменили конечную сумму из не более чем $\log_4 n + 1$ членов на сумму бесконечного ряда, а также переписали $3^{\log_4 n}$ как $n^{\log_4 3}$, что есть $o(n)$, так как $\log_4 3 < 1$.)

Часто преобразование рекуррентного соотношения в сумму приводит к довольно сложным выкладкам. При этом важно следить за двумя вещами: сколько шагов подстановки требуется и какова сумма членов, получающихся на данном шаге. Иногда несколько первых шагов позволяют отгадать ответ, который затем удаётся доказать по индукции (с меньшим количеством вычислений).

Особенно много хлопот доставляют округления (переход к целой части). Для начала стоит предположить, что значения параметра таковы, что округления не требуются (в нашем примере при $n = 4^k$ ничего округлять не придётся). Вообще говоря, такое предположение не вполне законно, так как оценку надо доказать для всех достаточно больших целых чисел, а не только для степеней четвёрки. Мы увидим в разделе 4.3, как можно обойти эту трудность (см. также задачу 4-5).

Деревья рекурсии

Процесс подстановки соотношения в себя можно изобразить в виде **дерева рекурсии** (recursion tree). Как это делается, показано на рис. 4.1 на примере соотношения

$$T(n) = 2T(n/2) + n^2.$$

Для удобства предположим, что n — степень двойки. Двигаясь от (а) к (г), мы постепенно разворачиваем выражение для $T(n)$, используя выражения для $T(n)$, $T(n/2)$, $T(n/4)$ и т.д. Теперь мы можем вычислить $T(n)$, складывая значения вершин на каждом уровне. На верхнем уровне получаем n^2, на втором — $(n/2)^2 + (n/2)^2 = n^2/2$, на третьем — $(n/4)^2+(n/4)^2+(n/4)^2+(n/4)^2 = n^2/4$. Получается убывающая геометрическая прогрессия, сумма которой отличается от её первого члена не более чем на постоянный множитель. Итак, $T(n) = \Theta(n^2)$.

На рис. 4.2 показан более сложный пример — дерево для соотношения

$$T(n) = T(n/3) + T(2n/3) + n$$

Total=всего

Рисунок 4.1 Дерево рекурсии для соотношения $T(n) = 2T(n/2) + n^2$. Высота полностью развернутого дерева (г) равна $\lg n$ (дерево имеет $\lg n + 1$ уровней).

Рисунок 4.2 Дерево рекурсии для соотношения $T(n) = T(n/3) + T(2n/3) + n$.

(для простоты мы вновь игнорируем округления). Здесь сумма значений на каждом уровне равна n. Дерево обрывается, когда значения аргумента становятся сравнимыми с 1. Для разных ветвей это происходит на разной глубине, и самый длинный путь $n \to (2/3)n \to (2/3)^2 n \to \cdots \to 1$ требует около $k = \log_{3/2} n$ шагов (при таком k мы имеем $(2/3)^k n = 1$). Поэтому $T(n)$ можно оценить как $O(n \lg n)$.

Упражнения

4.2-1 Итерируя (подставляя в себя) соотношение $T(n) = 3T(\lfloor n/2 \rfloor) + n$, найдите хорошую верхнюю асимптотическую оценку для $T(n)$.

4.2-2 Анализируя дерево рекурсии, покажите, что $T(n) = T(n/3) + T(2n/3) + n$ влечёт $T(n) = \Omega(n \lg n)$.

4.2-3 Нарисуйте дерево рекурсии для $T(n) = 4T(\lfloor n/2 \rfloor) + n$ и получите асимптотически точные оценки для $T(n)$.

4.2-4 С помощью итераций решите соотношение $T(n) = T(n - a) + T(a) + n$, где $a \geqslant 1$ — некоторая константа.

4.2-5 С помощью дерева рекурсии решите соотношение $T(n) = T(\alpha n) + T((1 - \alpha)n) + n$, где α — константа в интервале $0 < \alpha < 1$.

4.3 Общий рецепт

Этот метод годится для рекуррентных соотношений вида

$$T(n) = aT(n/b) + f(n), \tag{4.5}$$

где $a \geqslant 1$ и $b > 1$ — некоторые константы, а f — положительная (по крайней мере для больших значений аргумента) функция. Он даёт общую формулу; запомнив её, можно решать в уме различные рекуррентные соотношения.

Соотношение (4.5) возникает, если алгоритм разбивает задачу размера n на a подзадач размера n/b, эти подзадачи решаются рекурсивно (каждая за время $T(n/b)$) и результаты объединяются. При этом затраты на разбиение и объединение описываются функцией $f(n)$ (в обозначениях раздела 1.3.2 $f(n) = C(n) + D(n)$). Например, для процедуры MERGE-SORT мы имеем $a = 2$, $b = 2$, $f(n) = \Theta(n)$.

Как всегда, в формуле (4.5) возникает проблема с округлением: n/b может не быть целым. Формально следовало бы заменить

$T(n/b)$ на $T(\lfloor n/b\rfloor)$ или $T(\lceil n/b\rceil)$. Оба варианта, как мы увидим в следующем разделе, приводят к одному и тому же ответу, и для простоты мы будем опускать округление в наших формулах.

Основная теорема о рекуррентных оценках

Теорема 4.1. *Пусть $a \geqslant 1$ и $b > 1$ — константы, $f(n)$ — функция, $T(n)$ определено при неотрицательных n формулой*

$$T(n) = aT(n/b) + f(n),$$

где под n/b понимается либо $\lceil n/b\rceil$, либо $\lfloor n/b\rfloor$. Тогда:

1. Если $f(n) = O(n^{\log_b a-\varepsilon})$ для некоторого $\varepsilon > 0$, то $T(n) = \Theta(n^{\log_b a})$.

2. Если $f(n) = \Theta(n^{\log_b a})$, то $T(n) = \Theta(n^{\log_b a}\lg n)$.

3. Если $f(n) = \Omega(n^{\log_b a+\varepsilon})$ для некоторого $\varepsilon > 0$ и если $af(n/b) \leqslant cf(n)$ для некоторой константы $c < 1$ и достаточно больших n, то $T(n) = \Theta(f(n))$.

В чём суть этой теоремы? В каждом из трёх случаев мы сравниваем $f(n)$ с $n^{\log_b a}$; если одна из этих функций растёт быстрее другой, то она и определяет порядок роста $T(n)$ (случаи 1 и 3). Если обе функции одного порядка (случай 2), то появляется дополнительный логарифмический множитель и ответом служит формула $\Theta(n^{\log_b a}\lg n) = \Theta(f(n)\lg n)$.

Отметим важные технические детали. В первом случае недостаточно, чтобы $f(n)$ была просто меньше, чем $n^{\log_b a}$: нам нужен *"зазор"* размера n^{ε} для некоторого $\varepsilon > 0$. Точно так же в третьем случае $f(n)$ должна быть больше $n^{\log_b a}$ с запасом, и к тому же удовлетворять условию *"регулярности"* $af(n/b) \leqslant cf(n)$; проверка последнего условия, как правило, не составляет труда.

Заметим, что три указанных случая не исчерпывают всех возможностей: может оказаться, например, что функция $f(n)$ меньше, чем $n^{\log_b a}$, но зазор недостаточно велик для того, чтобы воспользоваться первым утверждением теоремы. Аналогичная *"щель"* есть и между случаями 2 и 3. Наконец, функция может не обладать свойством регулярности.

Применения основной теоремы

Рассмотрим несколько примеров, где применение теоремы позволяет сразу же выписать ответ.

Для начала рассмотрим соотношение

$$T(n) = 9T(n/3) + n.$$

В этом случае $a = 9$, $b = 3$, $f(n) = n$, а $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$.

Поскольку $f(n) = O(n^{\log_3 9-\varepsilon})$ для $\varepsilon = 1$, мы применяем первое утверждение теоремы и заключаем, что $T(n) = \Theta(n^2)$.

Теперь рассмотрим соотношение

$$T(n) = T(2n/3) + 1.$$

Здесь $a = 1$, $b = 3/2$, $f(n) = 1$ и $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Подходит случай 2, поскольку $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, и мы получаем, что $T(n) = \Theta(\lg n)$.

Для соотношения

$$T(n) = 3T(n/4) + n\lg n$$

мы имеем $a = 3$, $b = 4$, $f(n) = n\lg n$; при этом $n^{\log_b a} = n^{\log_4 3} = O(n^{0{,}793})$. Зазор (с $\varepsilon \approx 0{,}2$) есть, остаётся проверить условие регулярности. Для достаточно большого n имеем $af(n/b) = 3(n/4)\lg(n/4) \leqslant (3/4)n\lg n = cf(n)$ для $c = 3/4$. Тем самым по третьему утверждению теоремы $T(n) = \Theta(n\lg n)$.

Вот пример, когда теорему применить не удаётся: пусть $T(n) = 2T(n/2) + n\lg n$. Здесь $a = 2$, $b = 2$, $f(n) = n\lg n$, $n^{\log_b a} = n$. Видно, что $f(n) = n\lg n$ асимптотически больше, чем $n^{\log_b a}$, но зазор недостаточен: отношение $f(n)/n^{\log_b a} = (n\lg n)/n = \lg n$ не оценивается снизу величиной n^ε ни для какого $\varepsilon > 0$. Это соотношение попадает в промежуток между случаями 2 и 3; для него можно получить ответ по формуле из упр. 4.4-2.

Упражнения

4.3-1 Используя основную теорему, найдите асимптотически точные оценки для соотношения

а. $T(n) = 4T(n/2) + n$;
б. $T(n) = 4T(n/2) + n^2$;
в. $T(n) = 4T(n/2) + n^3$.

4.3-2 Время работы алгоритма A описывается соотношением $T(n) = 7T(n/2) + n^2$, а время работы алгоритма A' — соотношением $T'(n) = aT'(n/4) + n^2$. При каком наибольшем целом значении a алгоритм A' асимптотически быстрее, чем A?

4.3-3 Используя основную теорему, покажите, что соотношение $T(n) = T(n/2) + \Theta(1)$ (для двоичного поиска, см. упр. 1.3-5) влечёт $T(n) = \Theta(\lg n)$.

4.3-4 Покажите, что условия регулярности (случай 3) не вытекает из других условий: приведите пример функции f, для которой существует требуемая константа ε, но условие регулярности не выполнено.

⋆ 4.4 Доказательство Теоремы 4.1

Мы приведём доказательство теоремы 4.1 для дотошных читателей; в дальнейшем оно не понадобится, так что при первом чтении его вполне можно пропустить.

Доказательство состоит из двух частей. Сначала мы рассматриваем только те n, которые являются степенями числа b; все основные идеи видны уже для этого случая. Затем полученный результат распространяется на все натуральные числа, при этом мы аккуратно следим за округлениями и т. п.

В этом разделе мы позволим себе не совсем корректно обращаться с асимптотической записью: будем использовать её для функций, определённых только на степенях числа b, хотя определение требует, чтобы оценки доказывались для всех достаточно больших натуральных чисел.

Из контекста будет понятно, что имеется в виду, но нужно быть внимательным, чтобы не запутаться: если о функции $T(n)$ ничего не известно, то оценка $T(n) = O(n)$ для n, являющихся степенями двойки, ничего не гарантирует для произвольных n. (Возможно, что $T(n) = n$ при $n = 1, 2, 4, 8, \ldots$ и $T(n) = n^2$ при остальных n.)

4.4.1 Случай натуральных степеней

Пусть $T(n)$ определено для чисел, являющихся (натуральными) степенями числа $b > 1$ (не обязательно целого) и удовлетворяет соотношению (4.5), т.е.

$$T(n) = aT(n/b) + f(n).$$

Мы получим оценку для T так: перейдём от этого соотношения к суммированию (лемма 4.2), затем оценим полученную сумму (лемма 4.3) и подведём итоги (лемма 4.4).

Лемма 4.2. *Пусть* $a \geqslant 1$, $b > 1$ *— константы, и пусть* $f(n)$ *— неотрицательная функция, определённая на степенях* b. *Пусть* $T(n)$ *— функция, определённая на степенях* b *соотношением* $T(n) = aT(n/b) + f(n)$ *при* $n > 1$, *причём* $T(1) > 0$. *Тогда*

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j). \tag{4.6}$$

Рисунок 4.3 Дерево рекурсии для соотношения $T(n) = aT(n/b) + f(n)$ является полным a-ичным деревом высоты $\log_b n$ с $n^{\log_b a}$. Сумма весов по уровням показана справа, а общая сумма даётся формулой (4.6).

Доказательство. Последовательно подставляя соотношение само в себя, получаем

$$\begin{aligned} T(n) &= f(n) + aT(n/b) \\ &= f(n) + af(n/b) + a^2T(n/b^2) \\ &= f(n) + af(n/b) + a^2f(n/b^2) + \ldots \\ &\quad + a^{\log_b n - 1} f(n/b^{\log_b n - 1}) + a^{\log_b n} T(1). \end{aligned}$$

Поскольку $a^{\log_b n} = n^{\log_b a}$, последний член может быть записан как $\Theta(n^{\log_b a})$. Оставшиеся члены образуют сумму, фигурирующую в утверждении леммы. □

Дерево рекурсии

Доказательство леммы 4.2 можно пояснить в терминах дерева рекурсии (рис. 4.3), если число a целое (хотя само доказательство этого не требует). В корне стоит число $f(n)$, в каждом из a его детей стоит число $f(n/b)$, в каждом из a^2 внуков стоит $f(n/b^2)$ и т. д. На уровне j имеется a^j вершин; вес каждой — $f(n/b^j)$. Листья находятся на расстоянии $\log_b n$ от корня, и имеют положительный вес $T(1)$; всего на дереве $a^{\log_b n} = n^{\log_b a}$ листьев.

Равенство (4.6) получается, если сложить веса на всех уровнях: общий вес на j-м уровне есть $a^j f(n/b^j)$, а общий вес внутренней части дерева есть

$$\sum_{j=0}^{\log_b n-1} a^j f(n/b^j).$$

Если рекуррентное соотношение произошло из алгоритма типа *"разделяй и властвуй"*, эта сумма отражает стоимость разбиения задачи на подзадачи и объединения решений. Суммарный вес всех листьев есть стоимость решения всех $n^{\log_b a}$ задач размера 1, которая составляет $\Theta(n^{\log_b a})$.

В терминах дерева рекурсии легко объяснить, чему соответствуют три случая в формулировке основной теоремы. В первом случае основная часть веса сосредоточена в листьях, в третьем — в корне, во втором вес равномерно распределён по уровням дерева.

Теперь оценим величину суммы в формуле (4.6).

Лемма 4.3. *Пусть $a \geqslant 1$, $b > 1$ — константы, $f(n)$ — неотрицательная функция, определённая на натуральных степенях b. Рассмотрим функцию $g(n)$, определённую формулой*

$$g(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \tag{4.7}$$

(для n, являющихся степенями b). Тогда

1. Если $f(n) = O(n^{\log_b a-\varepsilon})$ для некоторой константы $\varepsilon > 0$, то $g(n) = O(n^{\log_b a})$.

2. Если $f(n) = \Theta(n^{\log_b a})$, то $g(n) = \Theta(n^{\log_b a} \lg n)$.

3. Если $af(n/b) \leqslant cf(n)$ для некоторой константы $c < 1$ и для всех $n \geqslant b$, то $g(n) = \Theta(f(n))$.

Доказательство. 1. В первом случае достаточно доказать утверждение леммы для функции $f(n) = n^\alpha$, где $\alpha = \log_b a - \varepsilon$ Для такой функции f равенство (4.7) может быть переписано так: (теперь $\alpha = \log_b a$)

$$\begin{aligned} g(n) &= f(n) + af(n/b) + a^2 f(n/b^2) + \ldots + a^{k-1} f(n/b^{k-1}) \\ &= n^\alpha + a(n/b)^\alpha + a^2(n/b^2)^\alpha + \ldots + a^{k-1}(n/b^{k-1})^\alpha; \end{aligned} \tag{4.8}$$

правая часть представляет собой геометрическую прогрессию длины $k = \log_b n$ со знаменателем a/b^α; этот знаменатель больше 1, так как $\alpha < \log_b a$ и $b^\alpha < a$. Для такой прогресии сумма по порядку равна последнему члену (отличается от него не более чем на константу раз). Этот последний член есть $O(a^k) = O(n^{\log_b a})$.

Для первого случая утверждение леммы доказано.

2. Во втором случае аналогичное рассуждение даёт сумму того же вида:

$$\begin{aligned} g(n) &= f(n) + af(n/b) + a^2 f(n/b^2) + \ldots \\ &= n^\alpha + a(n/b)^\alpha + a^2(n/b^2)^\alpha + \ldots, \end{aligned} \tag{4.9}$$

но теперь $\alpha = \log_b a$ и потому знаменатель геометрической прогрессии равен 1 и все её члены равны. Их число есть $\log_b n$, и потому сумма равна

$$n^{\log_b a} \log_b n = \Theta(n^{\log_b a} \lg n).$$

Случай 2 разобран.

3. В этом случае условие регулярности функции f гарантирует, что в нашей сумме каждый следующий член не превосходит предыдущего, умноженного на $c < 1$. Тем самым её можно оценить сверху убывающей геометрической прогрессией со знаменателем c, и сумма такой прогрессии не более чем в константу (равную $1/(1-c)$) раз превосходит первый член (но и не меньше его, так как все слагаемые неотрицательны). Таким образом, $g(n) = \Theta(f(n))$ для n, являющихся степенями b. Доказательство леммы завершено. □

Теперь мы можем доказать основную теорему о рекуррентных оценках для случая, когда n есть натуральная степень b.

Лемма 4.4. *Пусть $a \geqslant 1$, $b > 1$ — константы, и пусть $f(n)$ — неотрицательная функция, определённая на степенях b. Пусть $T(n)$ — функция, определённая на степенях b соотношением $T(n) = aT(n/b) + f(n)$ при $n > 1$ и $T(1) > 0$. Тогда:*

1. Если $f(n) = O(n^{\log_b a - \varepsilon})$ для некоторого $\varepsilon > 0$, то $T(n) = \Theta(n^{\log_b a})$.

2. Если $f(n) = \Theta(n^{\log_b a})$, то $T(n) = \Theta(n^{\log_b a}) \lg n$.

3. Если $f(n) = \Omega(n^{\log_b a + \varepsilon})$ для некоторого $\varepsilon > 0$ и если $af(n/b) \leqslant cf(n)$ для некоторой константы $c < 1$ и для достаточно больших n, то $T(n) = \Theta(f(n))$.

??????? Как убрать точку в конце слова Доказательство?

Доказательство. состоит в комбинации лемм 4.2 и 4.3. В первом случае получаем

$$T(n) = \Theta(n^{\log_b a}) + O(n^{\log_b a}) = \Theta(n^{\log_b a}).$$

Во втором случае имеем

$$T(n) = \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \lg n) = \Theta(n^{\log_b a} \lg n).$$

В третьем случае

$$T(n) = \Theta(n^{\log_b a}) + \Theta(f(n)) = \Theta(f(n)).$$

Заметим, что в последнем случае условие "$f(n) = \Omega(n^{\log_b a+\varepsilon})$ *для некоторого* $\varepsilon > 0$" можно было бы опустить, так как оно вытекает из условия регулярности (см. упр. 4.4-3). □

Лемма 4.4 доказана.

4.4.2 Целые приближения сверху и снизу

Нам осталось разобраться подробно с произвольными n, не являющимися степенями числа b. В этом случае n/b подлежит округлению и соотношение имеет вид

$$T(n) = aT(\lceil n/b \rceil) + f(n) \tag{4.10}$$

или

$$T(n) = aT(\lfloor n/b \rfloor) + f(n). \tag{4.11}$$

Надо убедиться, что оценка остаётся в силе и для этого случая.

Посмотрим, какие изменения надо внести в наши рассуждения. Вместо последовательности $n, n/b, n/b^2, \ldots$ теперь надо рассматривать последовательность n_i, определённую так:

$$n_i = \begin{cases} n, & \text{если } i = 0, \\ \lceil n_{i-1}/b \rceil, & \text{если } i > 0. \end{cases} \tag{4.12}$$

(мы рассматриваем случай округления с избытком). Эта последовательность — убывающая, но не обязательно стремится к единице. Однако мы можем утверждать, что после $\log_b n$ итераций получится число, ограниченное не зависящей от n (хотя зависящей от b) константой.

В самом деле, из неравенства $\lceil x \rceil \leqslant x + 1$ следует, что

$$\begin{aligned} n_0 &\leqslant n, \\ n_1 &\leqslant \frac{n}{b} + 1, \\ n_2 &\leqslant \frac{n}{b^2} + \frac{1}{b} + 1, \\ n_3 &\leqslant \frac{n}{b^3} + \frac{1}{b^2} + \frac{1}{b} + 1, \\ &\ldots\ldots\ldots\ldots\ldots\ldots\ldots \end{aligned}$$

Поскольку $1 + 1/b + 1/b^2 + \ldots \leqslant 1/(b-1)$, для $i = \lfloor \log_b n \rfloor$ мы получаем $n_i \leqslant n/b^i + b/(b-1) \leqslant b + b/(b-1) = O(1)$.

Итерируя соотношение (4.10), мы получаем

$$\begin{aligned}
T(n) &= f(n_0) + aT(n_1)\\
&= f(n_0) + af(n_1) + a^2T(n_2)\\
&\leqslant f(n_0) + af(n_1) + a^2f(n_2) + \ldots +\\
&\quad + a^{\lfloor \log_b n\rfloor - 1} f(n_{\lfloor \log_b n\rfloor - 1}) + a^{\lfloor \log_b n\rfloor} T(n_{\lfloor \log_b n\rfloor})\\
&= \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n\rfloor - 1} a^j f(n_j). \qquad (4.13)
\end{aligned}$$

Это выражение очень похоже на (4.6), только здесь n не обязано быть степенью числа b.

[Строго говоря, сказанное требует некоторых уточнений. Дело в том, что величина $T(n_{\lceil \log bn\rceil})$ может оказаться равной нулю. Но мы знаем, что $f(n)$ асимптотически положительна, т.е. положительна для всех n, начиная с некоторого n_0. Поэтому в развёртывании суммы нужно остановиться, немного не дойдя до этого n_0. Мы опускаем подробности.]

Теперь надо разобраться с соотношениями между слагаемыми в сумме

$$g(n) = \sum_{j=0}^{\lfloor \log_b n\rfloor - 1} a^j f(n_j). \qquad (4.14)$$

Раньше мы сравнивали эту сумму с геометрической прогрессией, в которой знаменатель был больше единицы (первый случай), равен единице (второй случай) или меньше единицы (третий случай). Третий случай не вызывает дополнительных проблем, так как в условии регулярности также предусматривается округление (причём в ту же сторону, что и в рекуррентном соотношении). Во втором случае мы должны оценить, насколько велики изменения вследствие замены n/b^i на n_i. Как мы видели, разница не превосходит $b/(b-1)$, но надо ещё от абсолютной ошибки перейти к относительной. Нам надо проверить, что $f(n_j) = O(n^{\log_b a}/a^j) = O((n/b^j)^{\log_b a})$, тогда проходит доказательство леммы 4.3 (случай 2). Заметим, что $b^j/n \leqslant 1$ при $j \leqslant \lfloor \log_b n\rfloor$. Из оценки $f(n) = O(n^{\log_b a})$ вытекает, что для подходящего c и достаточно больших n_j

$$\begin{aligned}
f(n_j) &\leqslant c\left(\frac{n}{b^j}+\frac{b}{b-1}\right)^{\log_b a}\\
&= c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1+\left(\frac{b^j}{n}\cdot\frac{b}{b-1}\right)\right)^{\log_b a}\\
&\leqslant c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1+\frac{b}{b-1}\right)^{\log_b a}\\
&= O\left(\frac{n^{\log_b a}}{a^j}\right).
\end{aligned}$$

Последний переход использует то, что $c(1+b/(b-1))^{\log_b a}$ — константа. Итак, случай 2 разобран.

Рассуждение для случая 1 почти такое же. Там нужно доказать оценку $f(n_j) = O(n^{\log_b a-\varepsilon})$; а это делается примерно так же, как и в случае 2, хотя преобразования будут несколько сложнее.

Итак, мы рассмотрели случай произвольного n для округления с избытком. Аналогично можно рассмотреть случай округления с недостатком, при этом нужно будет доказывать, что n_j не может быть сильно меньше n/b^j.

Упражнения

4.4-1* Укажите простую явную формулу для n_i из (4.12), если b — положительное целое число.

4.4-2* Покажите, что если $f(n) = \Theta(n^{\log_b a}\lg^k n)$, где $k \geqslant 0$, то соотношение (4.5) влечёт $T(n) = \Theta(n^{\log_b a}\lg^{k+1} n)$. Для простоты рассмотрите лишь случай целых степеней b.

4.4-3* Покажите, что в случае 3 основной теоремы одно из условий лишнее: условие регулярности ($af(n/b) \leqslant cf(n)$ для некоторого $c < 1$) гарантирует, что существует $\varepsilon > 0$, для которого $f(n) = \Omega(n^{\log_b a+\varepsilon})$.

Задачи

4-1 Примеры рекуррентных соотношений

Дайте как можно более точные асимптотические верхние и нижние оценки для следующих соотношений (считаем, что $T(n)$ — константа при $n \leqslant 2$):

а. $T(n) = 2T(n/2) + n^3$.

б. $T(n) = T(9n/10) + n$.

в. $T(n) = 16T(n/4) + n^2$.
г. $T(n) = 7T(n/3) + n^2$.
д. $T(n) = 7T(n/2) + n^2$.
е. $T(n) = 2T(n/4) + \sqrt{n}$.
ж. $T(n) = T(n-1) + n$.
з. $T(n) = T(\sqrt{n}) + 1$.

4-2 Недостающее число

Массив $A[1\,..\,n]$ содержит все целые числа от 0 до n, кроме одного. Чтобы найти пропущенное число, можно отмечать во вспомогательном массиве $B[0\,..\,n]$ все числа, встречающиеся в A, и затем посмотреть, какое число не отмечено (всё вместе требует времени $O(n)$). При этом использование элемента массива A в качестве индекса считается элементарной операцией.

Наложим такое дополнительное ограничение на доступ к массиву A: за одно действие мы можем посмотреть заданный бит заданного элемента массива A.

Покажите, что и при таких ограничениях мы можем найти пропущенное число за время $O(n)$.

4-3 Время передачи параметров

В этой книге мы предполагаем, что передача параметров занимает постоянное время, даже если передаваемый параметр — массив. В большинстве языков программирования это так и есть, поскольку передаётся не сам массив, а указатель на него. Но возможны и другие варианты. Сравним три способа передачи массивов в качестве параметров:

1. Массив передается как указатель за время $\Theta(1)$.
2. Массив копируется за время $\Theta(N)$, где N — размер массива.
3. В процедуру передаётся только та часть массива, которая будет в ней использоваться. При этом требуется время $\Theta(q - p + 1)$, если передаётся участок $A[p\,..\,q]$.

а. Рассмотрим рекурсивный алгоритм двоичного поиска числа в упорядоченном массиве (упр. 1.3-5). Каково будет время его работы при каждом из указанных способов передачи параметра? (Аргументами рекурсивной процедуры являются искомый элемент и массив, в котором осуществляется поиск.)

б. Проведите такой анализ для алгоритма MERGE-SORT из раздела 1.3.1.

4-4 Ещё несколько рекуррентных соотношений

Укажите возможно более точные асимптотические верхние и нижние оценки для $T(n)$ в каждом из указанных ниже примеров. Предполагается, что $T(n)$ — константа при $n \leqslant 8$.

а. $T(n) = 3T(n/2) + n \lg n$.

б. $T(n) = 3T(n/3 + 5) + n/2$.
в. $T(n) = 2T(n/2) + n/\lg n$.
г. $T(n) = T(n-1) + 1/n$.
д. $T(n) = T(n-1) + \lg n$.
е. $T(n) = \sqrt{n}T(\sqrt{n}) + n$.

4-5 Переход от степеней к произвольным аргументам

Пусть мы получили оценку для $T(n)$ при всех n, являющихся степенями некоторого целого b. Как распространить её на произвольные n?

а. Пусть $T(n)$ и $h(n)$ — монотонно возрастающие функции, определённые для произвольных положительных аргументов (не обязательно целых), причём $T(n) \leqslant h(n)$ для всех n, являющихся степенями целого числа $b > 1$. Пусть известно, кроме того, что h *"растёт достаточно медленно"*: $h(n) = O(h(n/b))$. Докажите, что $T(n) = O(h(n))$.

б. Пусть для функции T выполнено соотношение $T(n) = aT(n/b) + f(n)$ при $n > n_0$; пусть $a \geqslant 1$, $b > 1$ и $f(n)$ монотонно возрастает. Пусть $T(n)$ монотонно возрастает при $n \leqslant n_0$, и при этом $T(n_0) \leqslant aT(n_0/b) + f(n_0)$. Докажите, что $T(n)$ монотонно возрастает.

в. Упростите доказательство теоремы 4.1 для случая монотонной и *"достаточно медленно растущей"* функции $f(n)$. Воспользуйтесь леммой 4.4.

4-6 Числа Фибоначчи

Числа Фибоначчи определяются соотношением (2.13). Здесь мы рассмотрим некоторые их свойства. Для этого определим **производящую функцию** (generating function), определяемую как **формальный степенной ряд** (formal power series)

$$\mathcal{F} = \sum_{i=0}^{\infty} F_i z^i = 0 + z + z^2 + 2z^3 + 3z^4 + 5z^5 + 8z^6 + 13z^7 + \ldots$$

а. Покажите, что $\mathcal{F}(z) = z + z\mathcal{F}(z) + z^2\mathcal{F}(z)$.

б. Покажите, что

$$\mathcal{F}(z) = \frac{z}{1 - z - z^2} = \frac{z}{(1 - \varphi z)(1 - \widehat{\varphi} z)} = \frac{1}{\sqrt{5}}\left(\frac{1}{1 - \varphi z} - \frac{1}{1 - \widehat{\varphi} z}\right)$$

где

$$\varphi = \frac{1 + \sqrt{5}}{2} = 1{,}61803\ldots$$

и

$$\widehat{\varphi} = \frac{1 - \sqrt{5}}{2} = -0{,}61803\ldots$$

в. Покажите, что

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}} (\varphi^i - \widehat{\varphi}^i) z^i.$$

г. Докажите, что F_i при $i > 0$ равно ближайшему к $\varphi^i/\sqrt{5}$ целому числу. (Указание: $|\widehat{\varphi}| < 1$.)

д. Докажите, что $F_{i+2} \geqslant \varphi^i$ для всех $i \geqslant 0$.

4-7 Тестирование микросхем

Имеется n одинаковых микросхем, способных проверять друг друга; некоторые из них исправны, некоторые — нет. Для проверки пара микросхем вставляется в специальную плату, после чего каждая из них сообщает о состоянии соседа. Исправная микросхема при этом никогда не ошибается, а неисправная может дать любой ответ.

Ответ A	Ответ B	Результат
B исправна	A исправна	обе хорошие или обе плохие
B исправна	A неисправна	хотя бы одна неисправна
B неисправна	A исправна	хотя бы одна неисправна
B неисправна	A неисправна	хотя бы одна неисправна

а. Покажите, что если больше половины микросхем плохие, то попарное тестирование не позволит узнать наверняка, какие именно микросхемы плохи (они смогут нас обмануть).

б. Пусть нужно найти хотя бы одну хорошую микросхему из n штук, из которых больше половины исправных. Покажите, что достаточно $\lfloor n/2 \rfloor$ попарных тестов, чтобы свести задачу к аналогичной задаче половинного размера.

в. Пусть исправно больше половины микросхем. Покажите, что можно найти все хорошие микросхемы за $\Theta(n)$ попарных тестов. (Воспользуйтесь рекуррентным соотношением.)

[Эту задачу можно переформулировать в иных терминах: имеется n одинаковых на вид предметов, но на самом деле они относятся к нескольким категориям. Есть прибор, который по двум предметам проверяет, одинаковы ли они. Известно, что предметы некоторой категории составляют большинство. Надо найти представитель этого большинства за $O(n)$ сравнений. Помимо рекуррентного решения, эта задача имеет простое итеративное решение. Заведём коробку, в которой будем накапливать одинаковые предметы, а также урну, куда можно выкидывать предметы. Перекладываем непросмотренные элементы в коробку или урну, поддерживая такое свойство: среди невыкинутых искомые составляют большинство.]

Замечания

Числа Фибоначчи рассматривались Фибоначчи (L. Fibonacci) в 1202 году. Муавр (A. De Moivre) применил производящие функции для решения рекуррентных соотношений (см. задачу 4-6). Основная теорема о рекуррентных соотношениях заимствована из работы Бентли, Хакен и Сакса [26], где приводится более сильный результат (включающий результат упр. 4.4-2). Кнут [121] и Лю [140] показывают, как решать линейные рекуррентные соотношения с помощью производящих функций. Дополнительные сведения о решении рекуррентных соотношений приводят Пурдом и Браун [164].

1257

5 Множества

В этой главе мы напоминаем определения, терминологию, обозначения и основные свойства множеств, отношений, функций, графов и деревьев. Читатели, знакомые с ними, могут пропустить эту главу (обращаясь к ней по мере необходимости).

5.1 Множества

Множество (set) состоит из **элементов** (members, elements). Если объект x является элементом множества S, мы пишем $x \in S$ (читается *"x принадлежит S"*). Если x не принадлежит S, пишем $x \notin S$. Можно задать множество перечислением его элементов через запятые в фигурных скобках. Например, множество $S = \{1, 2, 3\}$ содержит элементы $1, 2, 3$ и только их. Число 2 является элементом этого множества, а число 4 — нет, так что $2 \in S$, $4 \notin S$. Множество не может содержать двух одинаковых элементов, и порядок элементов не фиксирован. Два множества A и B **равны** (are equal), если они состоят из одних и тех же элементов. В этом случае пишут $A = B$. Например, $\{1, 2, 3, 1\} = \{1, 2, 3\} = \{3, 2, 1\}$.

Для часто используемых множеств имеются специальные обозначения:

- $\varnothing$ обозначает **пустое множество** (empty set), не содержащее ни одного элемента.
- $\mathbb{Z}$ обозначает множество **целых чисел** (integers); $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$;
- $\mathbb{R}$ обозначает множество **вещественных (действительных) чисел** (real numbers);
- $\mathbb{N}$ обозначает множество **натуральных чисел** (natural numbers); $\mathbb{N} = \{0, 1, 2, \ldots\}$[1]

Если все элементы множества A являются элементами множе-

---

[1] Иногда нумерацию натуральных чисел начинают с 1 — раньше так было принято.

ства B, (из $x \in A$ следует $x \in B$), говорят, что A является **подмножеством** (subset) множества B и пишут $A \subseteq B$. Если при этом A не совпадает с B, то A называется **собственным подмножеством** (proper subset) множества B; в этом случае пишут $A \subset B$. (Многие авторы используют обозначение $A \subset B$ для подмножеств, а не для собственных подмножества.) Для любого множества A выполнено соотношение $A \subseteq A$. Множества A и B равны тогда и только тогда, когда $A \subseteq B$ и $B \subseteq A$. Для любых трёх множеств A, B и C из $A \subseteq B$ и $B \subseteq C$ следует $A \subseteq C$. Для любого множества A имеет место соотношение $\varnothing \subseteq A$.

Иногда множество определяется как часть другого множества: мы можем выделить из множества A все элементы, обладающие некоторым свойством, и образовать из них новое множество B. Например, множество чётных чисел можно определить как $\{x : x \in \mathbb{Z}$ и $x/2$ — целое число$\}$. Это обычно читают *"множество x из $\mathbb{Z}$, для которых..."* Иногда вместо двоеточия используется вертикальная черта.

Для любых множеств A и B можно построить следующие множества, получаемые с помощью **теоретико-множественных операций** (set operations):

- **Пересечение** (intersection) множеств A и B определяется как множество
$$A \cap B = \{x : x \in A \text{ и } x \in B\}.$$
- **Объединение** (union) множеств A и B определяется как множество
$$A \cup B = \{x : x \in A \text{ или } x \in B\}.$$
- **Разность** (difference) множеств A и B определяется как множество
$$A \setminus B = \{x : x \in A \text{ и } x \notin B\}.$$

Теоретико-множественные операции обладают следующими свойствами:

Свойства пустого множества (empty set laws):

$$A \cap \varnothing = \varnothing, \qquad A \cup \varnothing = A.$$

Идемпотентность (idempotency laws):

$$A \cap A = A, \qquad A \cup A = A.$$

Коммутативность (commutative laws):

$$A \cap B = B \cap A, \qquad A \cup B = B \cup A.$$

??? На картинке следует заменить минус на \

Рисунок 5.1 Диаграмма Венна, иллюстрирующая первый из законов де Моргана (5.2). Множества A, B, C изображены кругами на плоскости.

Ассоциативность (associative laws):

$$A \cap (B \cap C) = (A \cap B) \cap C, \qquad A \cup (B \cup C) = (A \cup B) \cup C.$$

Дистрибутивность (distributive laws):

$$\begin{aligned} A \cap (B \cup C) &= (A \cap B) \cup (A \cap C), \\ A \cup (B \cap C) &= (A \cup B) \cap (A \cup C) \end{aligned} \tag{5.1}$$

Законы поглощения (absorption laws):

$$A \cap (A \cup B) = A, \qquad A \cup (A \cap B) = A$$

Законы де Моргана (DeMorgan's laws):

$$\begin{aligned} A \setminus (B \cap C) &= (A \setminus B) \cup (A \setminus C), \\ A \setminus (B \cup C) &= (A \setminus B) \cap (A \setminus C) \end{aligned} \tag{5.2}$$

Рис. 5.1 иллюстрирует первый из законов де Моргана (5.1); множества A, B и C изображены в виде кругов на плоскости.

Часто все рассматриваемые множества являются подмножества некоторого фиксированного множества, называемого **универсумом** (universe). Например, если нас интересуют множества, элементами которых являются целые числа, то в качестве универсума можно взять множество $\mathbb{Z}$ целых чисел. Если универсум U фиксирован, можно определить **дополнение** (complement) множества A как $\overline{A} = U \setminus A$. Для любого $A \subseteq U$ верны такие утверждения:

$$\overline{\overline{A}} = A, \qquad A \cap \overline{A} = \varnothing, \qquad A \cup \overline{A} = U.$$

Из законов де Моргана (5.2) следует, что для любых множеств $A, B \subseteq U$ имеют место равенства

$$\overline{A \cap B} = \overline{A} \cup \overline{B}, \qquad \overline{A \cup B} = \overline{A} \cap \overline{B}.$$

Два множества A и B называются **непересекающимися** (disjoint), если они не имеют общих элементов, т.е. если $A \cap B = \varnothing$. Говорят, что семейство $\mathcal{S} = \{S_i\}$ непустых множеств образует **разбиение** (partition) множества S, если

- множества S_i **попарно не пересекаются** (are pairwise disjoint), т.е. $S_i \cap S_j = \varnothing$ при $i \neq j$,
- их объединение есть S, т.е.

$$S = \bigcup_{S_i \in \mathcal{S}} S_i$$

Другими словами, семейство $\mathcal{S}$ образует разбиение множества S, если любой элемент $s \in S$ принадлежит ровно одному из множеств S_i семейства.

Число элементов в множестве S называется его **мощностью** (cardinality), или **размером** (size), и обозначается $|S|$. Два множества имеют одну и ту же мощность, если между их элементами можно установить взаимно однозначное соответствие. Мощность пустого множества равна нулю: $|\varnothing| = 0$. Мощность **конечного** (finite) множества — натуральное число; для **бесконечных** (infinite) множеств понятие мощности требует аккуратного определения. Оно нам не понадобится; упомянем лишь, что множества, элементы которых можно поставить во взаимно однозначное соответствие с натуральными числами, называются **счётными** (countably infinite); бесконечные множества, не являющиеся счётными, называют **несчётными** (uncountable). Множество целых чисел $\mathbb{Z}$ счётно, в то время как множество вещественных чисел $\mathbb{R}$ несчётно.

Для любых двух конечных множеств A и B выполнено равенство

$$|A \cup B| = |A| + |B| - |A \cap B| \tag{5.3}$$

из этого равенства вытекает, что

$$|A \cup B| \leqslant |A| + |B|$$

Если множества A и B не пересекаются, то $|A \cap B| = 0$ и это неравенство обращается в равенство: $|A \cup B| = |A| + |B|$. Если $A \subseteq B$, то $|A| \leqslant |B|$.

Конечное множество из n элементов называют **n-элементным** (n-set); одноэлементное множество именуют иногда **синглетоном** (singleton). В английской литературе употребляется также термин k-subset, означающий k-элементное подмножество (какого-либо множества).

Для данного множества S можно рассмотреть множество всех его подмножеств, включая пустое множество и само S; его обозначают 2^S и называют **множеством-степенью** (power set). Например, $2^{\{a,b\}} = \{\varnothing, \{a\}, \{b\}, \{a,b\}\}$. Для конечного S множество 2^S содержит $2^{|S|}$ элементов.

Упорядоченная пара из двух элементов a и b обозначается (a,b) и формально может быть определена как $(a,b) = \{a, \{a,b\}\}$, так что (a,b) отличается от (b,a). [Это определение упорядоченной пары предложено Куратовским. Можно было бы использовать и другое определение, важно только, чтобы $(a, b) = (c, d)$ было равносильно $(a = c)$ и $(b = d)$.]

Декартово произведение (cartesian product) двух множеств A и B определяется как множество всех упорядоченных пар, у которых первый элемент принадлежит A, а второй — B. Обозначение: $A \times B$. Формально можно записать

$$A \times B = \{(a, b) : a \in A \text{ и } b \in B\}$$

Например, $\{a, b\} \times \{a, b, c\} = \{(a, a), (a, b), (a, c), (b, a), (b, b), (b, c)\}$. Для конечных множеств A и B мощность их произведения равна произведению мощностей:

$$|A \times B| = |A| \cdot |B|. \tag{5.4}$$

Декартово произведение n множеств $A_1, A_2, \ldots, A_n$ определяется как множество **n-ок** (n-tuples)

$$A_1 \times A_2 \times \ldots \times A_n = \{(a_1, a_2, \ldots, a_n) : a_i \in A_i \text{ при всех } i = 1, 2, \ldots, n\}$$

(формально можно определить тройку (a, b, c) как $((a, b), c)$, чётверку (a, b, c, d) как $((a, b, c), d)$ и так далее).

Число элементов в декартовом произведение равно произведению мощностей сомножителей:

$$|A_1 \times A_2 \times \ldots \times A_n| = |A_1| \cdot |A_2| \cdot \ldots \cdot |A_n|$$

Можно определить также **декартову степень**

$$A^n = A \times A \times \ldots \times A$$

как произведение n одинаковых сомножителей; для конечного A мощность A^n равна $|A|^n$. Отметим, что n-ки можно рассматривать как конечные последовательности длины n (см. с. ??).

Упражнения

5.1-1 Нарисуйте диаграмму Венна для первого из свойств дистрибутивности (5.1).

5.1-2 Докажите обобщение законов де Моргана на случай большего числа множеств:

$$\overline{A_1 \cap A_2 \cap \ldots \cap A_n} = \overline{A_1} \cup \overline{A_2} \cup \ldots \cup \overline{A_n},$$
$$\overline{A_1 \cup A_2 \cup \ldots \cup A_n} = \overline{A_1} \cap \overline{A_2} \cap \ldots \cap \overline{A_n}.$$

5.1-3 Докажите обобщение равенства (5.3), называемое **формулой включений и исключений** (principle of inclusion and exclusion):

$$\begin{aligned}|A_1 \cup A_2 \cup \ldots \cup A_n| = {} & |A_1| + |A_2| + \ldots + |A_n| \\ & - |A_1 \cap A_2| - |A_1 \cap A_3| - \ldots \\ & + |A_1 \cap A_2 \cap A_3| + \ldots \\ & + (-1)^{n-1}|A_1 \cap A_2 \cap \ldots \cap A_n|\end{aligned}$$

5.1-4 Доказать, что множество нечётных натуральных чисел счётно.

5.1-5 Доказать, что если множество S конечно и содержит n элементов, то множество-степень 2^S содержит 2^n элементов. (Другими словами, множество S имеет 2^n различных подмножеств.)

5.1-6 Дайте формально индуктивное определение n-ки, используя понятие упорядоченной пары.

5.2 Отношения

Бинарным отношением (binary relation) R между элементами множеств A и B называется подмножество декартова произведения $A \times B$. Если $(a, b) \in R$, пишут aRb и говорят, что элемент a находится в отношении R с элементом b. Бинарным отношением на множестве A называют подмножество декартова квадрата $A \times A$. Например, отношение *"быть меньше"* на множестве натуральных чисел есть множество $\{(a, b) : a, b \in \mathbb{N} \text{ и } a < b\}$. Под **$n$-местным отношением** ($n$-ary relation) на множествах $A_1, A_2, \ldots, A_n$ понимают подмножество декартова произведения $A_1 \times A_2 \times \ldots \times A_n$.

Бинарное отношение $R \subseteq A \times A$ называют **рефлексивным** (reflexive), если

$$aRa$$

для всех $a \in A$. Например, отношения "$=$" и "$\leqslant$" являются рефлексивными отношениями на множестве $\mathbb{N}$, но отношение "$<$" таковым не является. Отношение R называется **симметричным** (symmetric), если

$$aRb \text{ влечёт } bRa$$

для всех $a, b \in A$. Отношение равенства является симметричным, а отношения "$<$" и "$\leqslant$" — нет. Отношение R называют **транзитивным** (transitive), если

$$aRb \text{ и } bRc \text{ влечёт } aRc$$

для всех $a, b, c \in A$. Например, отношения $"<"$, $"\leqslant"$ и $"="$ являются транзитивными, а отношение $R = \{(a, b) : a, b \in \mathbb{N}$ и $a = b - 1\}$ — нет, так как $3R4$ и $4R5$, но не $3R5$.

Отношение, являющееся одновременно рефлексивным, симметричным и транзитивным, называют **отношением эквивалентности** (equivalence relation). Если R — отношение эквивалентности на множестве A, то можно определить **класс эквивалентности** (equivalence class) элемента $a \in A$ как множество $[a] = \{b \in A : aRb\}$ всех элементов, эквивалентных a. Например, на множестве натуральных чисел можно определить отношение эквивалентности, считая числа a и b эквивалентными, если их сумма $a + b$ чётна. Это отношение (назовём его R) действительно будет отношением эквивалентности. В самом деле, сумма $a + a$ всегда чётна, так что оно рефлексивно; $a + b = b + a$, так что R симметрично; наконец, если $a+b$ и $b+c$ — чётные числа, то $a+c = (a+b)+(b+c)-2b$ также чётно, так что R транзитивно. Класс эквивалентности числа 4 есть $[4] = \{0, 2, 4, 6, \ldots\}$, а класс эквивалентности числа 3 есть $[3] = \{1, 3, 5, 7, \ldots\}$. Основное свойство классов эквивалентности состоит в следующем:

Теорема 5.1 (Отношения эквивалентности соответствуют разбиениям). *Для любого отношения эквивалентности на множестве A классы эквивалентности образуют разбиение A. Напротив, для любого разбиения множества A отношение "быть в одном классе" является отношением эквивалентности.*

Доказательство. Чтобы доказать первое утверждение, надо показать, что классы эквивалентности непусты, попарно не пересекаются и в объединении дают всё множество A. По свойству рефлексивности $a \in [a]$, поэтому классы непусты и покрывают всё a. Докажем, что если классы $[a]$ и $[b]$ пересекаются, то они совпадают. Пусть c — их общий элемент, тогда aRc, bRc, cRb (симметричность) и aRb (транзитивность). Теперь видно, что $[b] \subseteq [a]$: если x — произвольный элемент b, то bRx и по транзитивности aRx. Аналогично, $[a] \subseteq [b]$ и потому $[a] = [b]$.

Второе утверждение теоремы совсем очевидно. □

Бинарное отношение R на множестве A называется **антисимметричным**, если

$$aRb \text{ и } bRa \text{ влечёт } a = b.$$

Например, отношение $"\leqslant"$ на натуральных числах является антисимметричным, поскольку из $a \leqslant b$ и $b \leqslant a$ следует $a = b$. Рефлексивное, антисимметричное и транзитивное отношение называется отношением **частичного порядка** (partial order), и множество вместе с таким отношением на нём называется **частично упорядоченным множеством** (partially ordered set). Например, отношение *"быть потом-*

ком" на множестве людей является частичным порядком, если мы считаем человека своим потомком.

Частично упорядоченное множество, даже конечное, может не иметь наибольшего элемента — такого элемента x, что yRx для любого элемента y. Следует различать понятия наибольшего и максимального элементов: элемент x называется **максимальным** (maximal), если не существует большего элемента, т.е. если из xRy следует $x = y$. Например, среди нескольких картонных коробок может не быть наибольшей (в которую помещается любая другая), но заведомо есть одна или несколько максимальных (которые не влезают ни в одну другую).

Частичный порядок называется **линейным** (total order, linear order), если для любых элементов a и b выполнено либо aRb, либо bRa (или оба — тогда они равны по свойству антисимметричности). Например, отношение "$\leqslant$" на множестве натуральных чисел является линейным порядком, а отношение *"быть потомком"* на множестве людей — нет (можно найти двух человек, не являющихся потомками друг друга).

Упражнения

5.2-1 Доказать, что отношение "$\subseteq$" на множестве всех подмножеств множества $\mathbb{Z}$ является отношением частичного, но не линейного порядка.

5.2-2 Показать, что для любого положительного n отношение $a \equiv b \pmod{n}$ является отношением эквивалентности на множестве $\mathbb{Z}$. (Говорят, что $a \equiv b \pmod{n}$, если существует целое q, для которого $a - b = qn$.) Сколько классов эквивалентности есть у этого отношения?

5.2-3 Приведите пример отношения, которое

а. рефлексивно и симметрично, но не транзитивно;

б. рефлексивно и транзитивно, но не симметрично;

в. симметрично и транзитивно, но не рефлексивно.

5.2-4 Пусть S — конечное множество, R — отношение эквивалентности на S. Докажите, что если R антисимметрично, что все классы эквивалентности содержат по одному элементу.

5.2-5 Профессор думает, что всякое симметричное и транзитивное отношение рефлексивно, и предлагает такое доказательство: из aRb следует bRa по симметричности, откуда следует aRa по транзитивности. Правильно ли это доказательство?

5.3 Функции

Пусть даны два множества A и B. **Функцией** (function), отображающей A в B, называется бинарное отношение $f \subseteq A \times B$, обладающее таким свойством: для каждого $a \in A$ существует ровно одно $b \in B$, для которого $(a, b) \in f$. Множество A называется **областью определения** (domain) функции; для множества B в русском языке нет общепринятого названия, а по-английски оно называется codomain.

Можно сказать, что функция f сопоставляет с каждым элементом множества A некоторый элемент множества B. Одному элементу множества A может соответствовать только один элемент множества B, хотя один и тот же элемент B может соответствовать нескольким различным элементам A. Например, бинарное отношение

$$f = \{(a, b) : a \in \mathbb{N} \text{ и } b = a \bmod 2\}$$

можно рассматривать как функцию $f\colon \mathbb{N} \to \{0, 1\}$, поскольку для каждого натурального числа a существует единственное $b \in \{0, 1\}$, равное $a \bmod 2$. Можно записать $f(0) = 0$, $f(1) = 1$, $f(2) = 0$ и так далее. С другой стороны, отношение

$$g = \{(a, b) : a, b \in \mathbb{N} \text{ и } a + b \text{ чётно}\}$$

не является функцией, поскольку (например) пары $(1, 3)$ и $(1, 5)$, принадлежащие этому отношению, имеют равные первые члены, но разные вторые.

Если пара (a, b) принадлежит отношению f, являющемуся функцией, то говорят, что b является **значением** (value) функции для **аргумента** (argument) a, и пишут $b = f(a)$. Чтобы задать функцию, надо указать её значение для каждого аргумента, принадлежащего её области определения. Например, можно задать функцию $f\colon \mathbb{N} \to \mathbb{N}$ формулой $f(n) = 2n$, которая означает, что $f = \{(n, 2n) : n \in \mathbb{N}\}$. Две функции $f, g\colon A \to B$ считаются **равными** (equal), если $f(a) = g(a)$ для всех $a \in A$. (Обратите внимание, что с формальной точки зрения мы считаем функции $f\colon A \to B_1$ и $g\colon A \to B_2$ различными при $B_1 \neq B_2$, даже если $f(a) = g(a)$ при всех a!)

Конечной последовательностью (finite sequence) длины n называют функцию f, область определения которой есть множество $\{0, 1, 2, \ldots, n-1\}$. Конечную последовательность часто записывают как список её значений, т.е. как $\langle f(0), f(1), \ldots, f(n-1)\rangle$. **Бесконечной последовательностью** (infinite sequence) называется функция, областью определения которой является множество $\mathbb{N}$ натуральных чисел. Например, последовательность Фибоначчи, заданная уравнением (2.13), может быть записана как $\langle 0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots\rangle$.

Если область определения функции является декартовым произведением нескольких множества, мы обычно опускаем дополнительные скобки вокруг аргументов. Например, если $f\colon A_1 \times A_2 \times \ldots \times A_n \to B$, мы пишем $f(a_1, a_2, \ldots, a_n)$ вместо более формального $f((a_1, a_2, \ldots, a_n))$. Каждое из a_i также называется **аргументом** (argument) функции f, хотя формально её аргументов следовало бы считать n-ку $(a_1, a_2, \ldots, a_n)$.

Если $b = f(a)$ для некоторой функции $f\colon A \to B$ и некоторых $a \in A$, $b \in B$, то элемент b называют **образом** (image) элемента a. Для произвольного подмножества A' множества A его образ $f(A')$ определяют формулой

$$f(A') = \{b \in B : b = f(a) \text{для некоторого } a \in A'\}.$$

Множество значений (range) функции f определяется как образ области её определения, т.е. как $f(A)$. Например, множество значений функции $f\colon \mathbb{N} \to \mathbb{N}$, определённой формулой $f(n) = 2n$, есть множество всех чётных натуральных чисел, которое можно записать как $f(\mathbb{N}) = \{m : m = 2n \text{ для некоторого } n \in \mathbb{N}\}$.

Функция $f\colon A \to B$ называется **сюръекцией** (surjection), или **наложением**, если её образ совпадает с множеством B, т.е. всякий элемент $b \in B$ является образом некоторого элемента $a \in A$. Например, функция $f\colon \mathbb{N} \to \mathbb{N}$, заданная формулой $f(n) = \lfloor n/2 \rfloor$, является сюръекцией. Функция $f(n) = 2n$ не будет сюръекцией, если считать, что $f\colon \mathbb{N} \to \mathbb{N}$, но будет таковой, если считать её отображающей множество натуральных чисел в множество чётных натуральных чисел. Функцию $f\colon A \to B$, являющуюся сюръекцией, называют также отображением A **на** B (onto B).

Функция $f\colon A \to B$ называется **инъекцией** (injection), или **вложением**, если различным аргументам соответствуют различные значения, т.е. если $f(a) \neq f(a')$ при $a \neq a'$. Например, функция $f(n) = 2n$ является инъекцией множества $\mathbb{N}$ в множество $\mathbb{N}$, поскольку любое число n является образом самое большее одного элемента ($n/2$, если n чётно; нечётные числа не являются образами никаких элементов). Функция $f(n) = \lfloor n/2 \rfloor$ не является инъекцией, так как (например) $f(2) = f(3) = 1$. В английской литературе для инъекций употребляется также термин *"one-to-one function"*.

Функция $f\colon A \to B$ называется **биекцией** (bijection), если она одновременно является инъекцией и сюръекцией. Например, функция $f(n) = (-1)^n \lceil n/2 \rceil$, рассматриваемая как функция, отображающая

$\mathbb{N}$ в $\mathbb{Z}$, является биекцией:

$$\begin{array}{rr} 0 \to & 0 \\ 1 \to & -1 \\ 2 \to & 1 \\ 3 \to & -2 \\ 4 \to & 2 \\ & \vdots \end{array}$$

Инъективность означает, что никакой элемент множества $\mathbb{Z}$ не является образом двух разных элементов множества $\mathbb{N}$. Сюръективность означает, что всякий элемент множества $\mathbb{Z}$ является образом хотя бы одного элемента множества $\mathbb{N}$. Биекции называют также **взаимно однозначными соответствиями** (one-to-one correspondence), поскольку они устанавливают соответствия между элементами множеств A и B. Биективная функция, отображающая множество A в себя, называется **перестановкой** (permutation) множества A.

Если функция f биективна, можно определить **обратную** (inverse) функцию f^{-1} соотношением

$$f^{-1}(b) = a \text{ тогда и только тогда, когда } f(a) = b.$$

Например, для рассмотренной выше функции $f(n) = (-1)^n \lceil n/2 \rceil$ обратная функция вычисляется по формуле

$$f^{-1}(m) = \begin{cases} 2m, & \text{если } m \geqslant 0, \\ -2m-1, & \text{если } m < 0. \end{cases}$$

Упражнения

5.3-1 Пусть A и B — конечные множества, и $f\colon A \to B$ — некоторая функция. Покажите, что

а. если f — инъекция, то $|A| \leqslant |B|$;

б. если f — сюръекция, то $|A| \geqslant |B|$.

5.3-2 Будет ли биекцией функция $f\colon \mathbb{N} \to \mathbb{N}$, заданная формулой $f(x) = x + 1$? Тот же вопрос для функции $\mathbb{Z} \to \mathbb{Z}$, заданной той же формулой.

5.3-3 Дайте определение обратного к бинарному отношению. (Если отношение является биекцией, то определение должно давать обратную биекцию в описанном выше смысле.)

5.3-4* Постройте биекцию $f\colon \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z}$.

Рисунок 5.2 Ориентированные и неориентированные графы. (а) Ориентированный граф (V,E), где $V = \{1,2,3,4,5,6\}$ и $E = \{(1,2),(2,2),(2,4),(2,5),(4,1),(4,5),(5,4),(6,3)\}$. Ребро $(2,2)$ является ребром-циклом. (б) Неориентированный граф $G = (V,E)$, где $V = \{1,2,3,4,5,6\}$ и $E = \{(1,2),(1,5),(2,5),(3,6)\}$. Вершина 4 является изолированной (не имеет смежных вершин). (в) Подграф графа (а), получающийся его ограничением на множество вершин $\{1,2,3,6\}$.

5.4 Графы

В этом разделе мы рассмотрим основные понятия, связанные с ориентированными и неориентированными графами. Следует иметь в виду, что терминология здесь не вполне устоялась и в разных книгах можно встретить разные определения, но по большей части различия невелики. Мы вернёмся к графам в главе 23, где рассматриваются различные алгоритмы на графах.

Ориентированный граф (directed graph) определяется как пара (V,E), где V — конечное множество, а E — бинарное отношение на V, т.е. подмножество множества $V \times V$. Ориентированный граф иногда для краткости называют **орграфом** (digraph). Множество V называют **множеством вершин** графа (vertex set); его элемент называют **вершиной** графа (vertex; множественное число vertices). Множество E называют **множеством рёбер** (edge set) графа; его элементы называют **рёбрами** (edges). На рисунке 5.2 (а) показан ориентированный граф с множеством вершин $\{1,2,3,4,5,6\}$. Вершины изображены кружками, а рёбра — стрелками. Заметим, что граф может содержать **рёбра-циклы** (self-loops), соединяющие вершину с собой.

В **неориентированном** (undirected) графе $G = (V,E)$ множество рёбер (V) состоит из **неупорядоченных** (unordered) пар вершин: парами являются множества $\{u,v\}$, где $u,v \in V$ и $u \neq v$. Мы будем обозначать неориентированное ребро как (u,v) вместо $\{u,v\}$; при этом для неориентированного графа (u,v) и (v,u) обозначают одно и то же ребро. Неориентированный граф не может содержать рёбер-циклов, и каждое ребро состоит из двух различных вершин (*”соединяя”* их). На рис. 5.2 (б) изображён неориентированный граф с множеством вершин $\{1,2,3,4,5,6\}$.

Многие понятия параллельно определяются для ориентированных и неориентированных графов (с соответствующими изменениями). Про ребро (u, v) ориентированного графа говорят, что оно **выходит из** (incident from, leaves) вершины u и **входит** (incident to, enters) в вершину v. Например, на рис. 5.2 (а) имеется три ребра, выходящих из вершины 2 $((2,2),(2,4),(2,5))$ и два ребра, в неё входящих $((1,2),(2,2))$. Про ребро (u, v) неориентированного графа говорят, что оно **инцидентно** вершинам (incident on vertices) u и v. Например, на рис. 2.5 (б) есть два ребра, инцидентные вершине 2 (рёбра $(1,2)$ и $(2,5)$).

Если в графе G имеется ребро (u, v), говорят, что вершина v **смежна** с вершиной u (is adjacent to u). Для неориентированных графов отношение смежности является симметричным, но для ориентированных графов это не обязательно. Если вершина v смежна с вершиной u в ориентированном графе, пишут $u \to v$. Для обоих рисунков 5.2 (а) и 5.2 (б) вершина 2 является смежной с вершиной 1, но лишь во втором из них вершина 1 смежна с вершиной 2 (в первом случае ребро $(2,1)$ отсутствует в графе).

Степенью (degree) вершины в неориентированном графе называется число инцидентных ей рёбер. Например, для графа рис. 5.2 (б) степень вершины 2 равна 2. Для ориентированного графа различают **исходящую степень** (out-degree), определяемую как число выходящих из неё рёбер, и **входящую степень** (in-degree), определяемую как число входящих в неё рёбер. Сумма исходящей и входящей степеней называется **степенью** (degree) вершины. Например, вершина 2 в графе рис. 5.2 (а) имеет входящую степень 2, исходящую степень 3 и степень 5.

Путь длины k (path of length k) из вершины u в вершину v определяется как последовательность вершин $\langle v_0, v_1, v_2, \ldots, v_k \rangle$, в которой $v_0 = u$, $v_k = v$ и $(v_{i-1}, v_i) \in E$ для всех $i = 1, 2, \ldots, k$. Таким образом, путь длины k состоит из k рёбер. Этот путь **содержит** (contains) вершины $v_0, v_1, \ldots, v_k$ и рёбра $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. Вершину v_0 называют **началом** пути, вершину v_k — его **концом**; говорят, что путь ведёт **из** v_0 **в** v_k. Если для данных вершин u и u' существует путь p из u в u', то говорят, что вершина u' **достижима из** u **по пути** p (u' is reachable from u via p). В этом случае мы пишем (для ориентированных графов) $u \overset{p}{\leadsto} u'$.

Путь называется **простым** (simple), если все вершины в нём различны. Например, на рис. 5.2 (а) есть простой путь $\langle 1, 2, 5, 4 \rangle$ длины 3, а также путь $\langle 2, 5, 4, 5 \rangle$ той же длины, не являющийся простым.

Подпуть (subpath) пути $p = \langle v_0, v_1, \ldots, v_k \rangle$ получится, если мы возьмём некоторое число идущих подряд вершин этого пути, т.е. последовательность $\langle v_i, v_{i+1}, \ldots, v_j \rangle$ при некоторых i, j, для которых $0 \leqslant i \leqslant j \leqslant k$.

Циклом (cycle) в ориентированном графе называется путь, в котором начальная вершина совпадает с конечной и который содержит хотя бы одно ребро. Цикл $\langle v_0, v_1, \ldots, v_k\rangle$ называется **простым**, если в нём нет одинаковых вершин (кроме первой и последней), т. е. если все вершины $v_1, v_2, \ldots, v_k$ различны. Ребро-цикл является циклом длины 1. Мы отождествляем циклы, отличающиеся сдвигом вдоль цикла: один и тот же цикл длины k может быть представлен k различными путями (в качестве начала и конца можно взять любую из k вершин). Например, на рис. 5.2 (а) пути $\langle 1, 2, 4, 1\rangle$, $\langle 2, 4, 1, 2\rangle$ и $\langle 4, 1, 2, 4\rangle$ представляют один и тот же цикл. Этот цикл является простым, в то время как цикл $\langle 1, 2, 4, 5, 4, 1\rangle$ таковым не является. На том же рисунке есть цикл $\langle 2, 2\rangle$, образованный единственным ребром-циклом $(2, 2)$. Ориентированный граф, не содержащий рёбер-циклов, называется **простым** (simple).

В неориентированном графе путь $\langle v_0, v_1, \ldots, v_k\rangle$ называется (простым) **циклом**, если $k \geqslant 3$, $v_0 = v_k$ и все вершины $v_1, v_2, \ldots, v_k$ различны. Например, на рис. 5.2 (б) имеется простой цикл $\langle 1, 2, 5, 1\rangle$.

Граф, в котором нет циклов, называется **ациклическим** (acyclic).

Неориентированный граф называется **связным** (connected), если для любой пары вершин существует путь из одной в другую. Для неориентированного графа отношение *"быть достижимым из"* является отношением эквивалентности на множестве вершин. Классы эквивалентности называются **связными компонентами** (connected components) графа. Например, на рис. 5.2 (б) имеются три связные компоненты: $\{1, 2, 5\}$, $\{3, 6\}$ и $\{4\}$. Неориентированный граф связен тогда и только тогда, когда он состоит из единственной связной компоненты.

Ориентированный граф называется **сильно связным** (strongly connected), если из любой его вершины достижима (по ориентированным путям) любая другая. Любой ориентированный граф можно разбить на **сильно связные компоненты** (strongly connected components), которые определяются как классы эквивалентности отношения *"u достижимо из v и v достижимо из u"*. Ориентированный граф сильно связен тогда и только тогда, когда состоит из единственной сильно связной компоненты. Граф рис. 5.2 (а) имеет три таких компоненты: $\{1, 2, 4, 5\}$, $\{3\}$ и $\{6\}$. Заметим, что вершины $\{3, 6\}$ не входят в одну сильно связную компоненту, так как 3 достижима из 6, но не наоборот.

Два графа $G = (V, E)$ и $G' = (V', E')$ называются **изоморфными** (isomorphic), если существует взаимно однозначное соответствие $f\colon V \to V'$ между множествами их вершин, при котором рёбрам одного графа соответствуют рёбра другого: $(u, v) \in E$ тогда и только тогда, когда $(f(u), f(v)) \in E'$. Можно сказать, что изоморфные графы — это один и тот же граф, в котором вершины названы по-разному. На рис. 5.3 (а) приведён пример двух изоморфных графов G и G' с множествами вершин $V = \{1, 2, 3, 4, 5, 6\}$ и

Рисунок 5.3 (а) Пара изоморфных графов. (б) Неизоморфные графы: верхний имеет вершину степени 4, а нижний — нет.

$V' = \{u, v, w, x, y, z\}$. Функция $f\colon V \to V'$, для которой $f(1) = u$, $f(2) = v$, $f(3) = w$, $f(4) = x$, $f(5) = y$, $f(6) = z$, является изоморфизмом. Напротив, графы на рис. 5.3 (б) не изоморфны, хотя оба имеют по 5 вершин и по 7 рёбер. Чтобы убедиться, что они не изоморфны, достаточно отметить, что в верхнем графе есть вершина степени 4, а в нижнем — нет.

Граф $G' = (E', V')$ называют **подграфом** (subgraph) графа $G = (E, V)$, если $E' \subseteq E$ и $V' \subseteq V$. Если в графе $G = (E, V)$ выбрать произвольное множество вершин V', то можно рассмотреть его подграф, состоящий из этих вершин и всех соединяющих их рёбер, т. е. граф $G' = (E', V')$, для которого

$$E' = \{(u, v) \in E : u, v \in V'\}$$

Этот подграф можно назвать **ограничением** графа G на множество вершин V' (subgraph of G induced by V'). Ограничение графа рис. 5.2 (а) на множество вершин $\{1, 2, 3, 6\}$ показано на рис. 5.2 (в) и имеет три ребра $(1, 2)$, $(2, 2)$, $(6, 3)$.

Для любого неориентированного графа G можно рассмотреть его **ориентированный вариант** (directed version), заменив каждое неориентированное ребро $\{u, v\}$ на пару ориентированных рёбер (u, v) и (v, u), идущих в противоположных направлениях. С другой стороны, для каждого ориентированного графа можно рассмотреть его **неориентированный вариант** (undirected version), забыв про ориентацию рёбер, удалив рёбра-циклы и соединив рёбра (u, v) и (v, u) в одно неориентированное ребро $\{u, v\}$. В ориентированном графе **соседом** (neighbor) вершины u называют любую вершину, соединённую с ней ребром (в ту или другую сторону); таким образом, v является соседом u тогда и только тогда, когда v смежно u или u смежно v. Для неориентированного графа выражения ”v — *сосед* u” и ”v *смежна с* u” являются синонимами.

Некоторые виды графов имеют специальные названия. **Полным** (complete) графом называют неориентированный граф, содержащий все возможные рёбра для данного множества вершин (любая вершина смежна любой другой). Неориентированный граф (V, E) называют **двудольным** (bipartite), если множество вершин V можно разбить на две часть V_1 и V_2 таким образом, что концы любого ребра оказываются в разных частях. Ациклический неориентированный граф называют **лесом** (forest), а связный ациклический неориентированный граф называют **деревом** (без выделенного корня; подробно деревья рассматриваются в следующем разделе). По-английски дерево без выделенного корня называется free tree. Ориентированный ациклический граф (directed acyclic graph) по-английски часто сокращают до *"dag"* (по первым буквам).

Иногда рассматривают обобщения понятия графа. Например, можно рассматривать **мультиграф** (multigraph), который похож на неориентированный граф, но может содержать много рёбер, соединяющих одну и ту же пару вершин, а также рёбра-циклы. **Гиперграф** (hypergraph) отличается от неориентированного графа тем, что он содержит **гиперрёбра** (hyperedges), соединяющие не две вершины, а произвольное множество вершин. Многие алгоритмы обработки обычных графов могут быть обобщены на такие графоподобные структуры.

Упражнения

5.4-1 На вечеринке каждый гость считает, сколько рукопожатий он сделал. Потом все числа складываются. Покажите, что получится чётное число, доказав следующую **лемму о рукопожатиях** (handshaking lemma): для неориентированного графа сумма степеней всех его вершин равна удвоенному числу рёбер.

5.4-2 Покажите, что требование $k \geqslant 3$ в определении цикла в неориентированном графе существенно (если его отменить, в любом графе, где есть хоть одно ребро, будет цикл).

5.4-3 Докажите, что если в графе (ориентированном или неориентированном) есть путь из вершины u в вершину v, то в нём есть простой путь из u в v. Докажите, что если в ориентированном графе есть цикл, то в нём есть простой цикл.

5.4-4 Покажите, что в связном неориентированном графе (V, E) число вершин превосходит число рёбер не более чем на 1: $|E| \geqslant |V| - 1$

5.4-5 Проверьте, что отношение *"быть достижимым из"* является (для неориентированного графа) отношением эквивалентности. Какие из трёх свойств, входящих в определение отношения эквивалентности, справедливы для отношения достижимости в ориентированном графе?

5.4-6 Нарисуйте неориентированную версию графа рис. 5.2 (а) и ориентированную версию графа рис. 5.2 (б).

5.4-7* Как можно представить гиперграф с помощью двудольного графа, изображая отношение инцидентности в гиперграфе отношением смежности в двудольном графе? (Указание: вершинами двудольного графа должны быть вершины гиперграфа, а также гиперрёбра гиперграфа.)

5.5 Деревья

Как и слово *"граф"*, слово *"дерево"* также употребляется в нескольких родственных смыслах. В этом разделе мы даём определения и рассматриваем свойства нескольких видов деревьев. В разделах 11.4 и 23.1 мы вернёмся к деревьям и рассмотрим способы их представления в программах.

5.5.1 Деревья без выделенного корня

Как мы говорили в разделе 5.4, **дерево** (без выделенного корня, free tree) определяется как связный ациклический неориентированный граф. Если неориентированный граф является ациклическим, но (возможно) несвязным, его называют **лесом** (forest); как и положено, лес состоит из деревьев (являющихся его связными компонентами). Многие алгоритмы обработки деревьев применимы и к лесам. На рис. 5.4 (а) изображено дерево; на рис. 5.4 (б) изображён лес. Лес рисунка 5.4 (б) не является деревом, так как не связен. Граф на рисунке 5.4 (в) не является ни деревом, ни даже лесом, так как в нём есть цикл.

В следующей теореме указано несколько важных свойств дере-

Рисунок 5.4 (а) Дерево без выделенного корня. (б) Лес. (в) Граф, содержащий цикл, не является ни деревом, ни лесом.

Рисунок 5.5 Два простых пути из u в v

вьев.

Теорема 5.2 (Свойства деревьев). *Пусть $G = (V, E)$ — неориентированный граф. Тогда следующие свойства равносильны:*

1. G является деревом (без выделенного корня).

2. Для любых двух вершин G существует единственный соединяющий их простой путь.

3. Граф G связен, но перестаёт быть связным, если удалить любое его ребро.

4. Граф G связен и $|E| = |V| - 1$.

5. Граф G ациклический и $|E| = |V| - 1$.

6. Граф G ациклический, но добавление любого ребра к нему порождает цикл.

Доказательство. (1) $\Rightarrow$ (2): Поскольку дерево связно, для любых двух вершин существует соединяющий их путь; выкинув из него лишнее, можем считать, что этот путь простой. Пусть есть два разных простых пути p_1 и p_2 из некоторый вершины u в другую вершину v (рис. 5.5). Посмотрим, где эти пути расходятся; пусть w — последняя общая вершина перед разветвлением, а x и y — различные вершины, следующие за w в путях p_1 и p_2. Двигаясь по первому пути, дождёмся момента, когда путь вновь пересекается со вторым в некоторой вершине z. Рассмотрим участок p' первого пути от w до z (через x) и участок второго пути p'' от w до z (через y). Пути p' и p'' не имеют общих вершин (не считая концов), поскольку z было первой вершиной на пути p_1 после w, попавшей в p_2. Поэтому мы получаем цикл (состоящий из p' и обращения пути p'').

(2) $\Rightarrow$ (3) Граф, очевидно, связен. Пусть (u, v) — любое его ребро. Если после его удаления граф останется связным, то в нём будет путь, соединяющий u и v — второй путь, что противоречит предположению.

(3) $\Rightarrow$ (4) По условию, граф связен, поэтому $|E| \geqslant |V| - 1$ (упр. 5.4-4). Докажем, что $|E| \leqslant |V| - 1$, рассуждая по индукции. Связный граф с $n = 1, 2$ вершинами имеет $n - 1$ рёбер. Пусть граф G имеет $n \geqslant 3$ рёбер и для графов с меньшим числом рёбер наше неравенство уже доказано. Удаление ребра разбивает граф G на $k \geqslant 2$ связных компонент (на самом деле на две, но это нам не важно). Для

каждой из компонент выполнено условие (3). Индуктивное предположение гарантирует, что общее число рёбер в них не больше $|V| - k \leqslant |V| - 2$. Значит, до удаления ребра было не более $|V| - 1$ рёбер.

(4) $\Rightarrow$ (5) Пусть граф связен и $|E| = |V| - 1$. Покажем, что граф не имеет циклов. Если цикл есть, удаление любого ребра из цикла не нарушает связности (можно пройти по остающейся части цикла). Будем повторять это до тех пор, пока не останется связный граф без циклов (дерево). Как мы уже знаем, для дерева число рёбер на единицу меньше числа вершин — но по предположению то же соотношение было и до удаления рёбер, так что удалить мы ничего не могли и с самого начала не было циклов.

(5) $\Rightarrow$ (6) Пусть граф G не имеет циклов и $|E| = |V| - 1$. Пусть G имеет k связных компонент; они по определению являются деревьями. Мы знаем, что в каждой из них число рёбер на единицу меньше числа вершин, поэтому общее число рёбер на k меньше числа вершин. Значит, $k = 1$ и граф представляет собой дерево. В нём любые две вершины могут быть соединены простым путём (мы уже знаем, что (1) $\Rightarrow$ (2)), и потому добавление любого ребра порождает цикл.

(6) $\Rightarrow$ (1) Пусть граф является ациклическим, но добавление любого ребра порождает цикл. Надо показать, что он связен. В самом деле, рассмотрим две произвольные вершины u и v. Мы знаем, что добавление ребра (u, v) порождает цикл. В этом цикле должно встречаться ребро (u, v), поскольку до его добавления цикла не было — но только один раз, и остальная часть цикла соединяет u и v, что и требовалось. □

5.5.2 Деревья с корнем. Ориентированные деревья

Дерево с корнем, или **корневое дерево** (rooted tree), получается, если в дереве (связном ациклическом неориентированном графе) выделить одну из вершин выделена, назвав её **корнем** (root). Вершины корневого дерева по-английски называются также *"nodes"*. На рисунке 5.6 (a) показано корневое дерево с 12 вершинами и корнем 7.

Пусть x — произвольная вершина корневого дерева с корнем в r. Существует единственный путь из r в x; все вершины, находящиеся на этом пути, мы называем **предками** (ancestors) вершины x. Если y является предком x, то x называется **потомком** (descendant) y. Каждую вершину мы считаем своим предком и потомком. Предки и потомки вершины x, не совпадающие с x, называются **собственными предками** (proper ancestors) и **собственными потомками** (proper descendants) вершины x. [Эта терминология не вполне удачна: фразу *"вершина является своим собственным потомком"* можно толковать по разному, и при одном толковании она высота = 4, глубина 0, глубина 1 и т.п.

Рисунок 5.6 (а) Корневое дерево высоты 4. Дерево нарисовано обычным образом: корень (вершина 7) нарисована сверху, соседние с ним вершины (дети корня, вершины глубины 1) нарисованы под ней, вершины следующего уровня (дети детей корня, вершины глубины 2) под ними и т. д. Для деревьев с порядком на детях для каждой вершины фиксирован порядок на множестве её детей. (б) То же самое корневое дерево, но с другим порядком (среди детей вершины 3), будет другим деревом с порядком на детях.

верна, а при другом нет.]

Для каждой вершины x можно рассмотреть дерево, состоящее из всех потомков x, в котором x считается корнем. Оно называется **поддеревом с корнем в** x (subtree rooted at x). Например, на рис. 5.6 (а) поддерево с корнем в 8 содержит вершины 8, 6, 5 и 9.

Если (y, x) — последнее ребро на пути из корня в x, то y называется **родителем** (parent) x, а x называется **ребёнком** y. [Раньше вместо *"родитель"* и *"ребёнок"* говорили *"отец"* (father) и *"сын"* (son), что было более логично, так как вообще-то у человека двое родителей, а у вершины дерева — не более одного. Зато принятая теперь в американской литературе терминология *"политически корректна"*.]

Корень является единственной вершиной, у которой нет родителя. Вершины, имеющие общего родителя, называют в американской литературе siblings; к сожалению, русского перевода у этого слова нет, и мы будем (как это делалось раньше) называть такие вершины **братьями**. Вершина корневого дерева, не имеющая детей, называется **листом** (leaf, external node). Вершины, имеющие детей, называются **внутренними** (internal).

Число детей у вершины корневого дерева называется её **степенью** (degree). Отметим, что для всех вершин, кроме корня, степень на единицу меньше степени той же вершины в том же дереве, если рассматривать дерево как неориентированный граф (поскольку тогда надо учитывать и ребро, идущее вверх). Длина пути от корня до произвольной вершины x называется **глубиной** (depth) вершины x. Максимальная глубина вершин дерева называется **высотой** (height) дерева.

Деревом с порядком на детях (ordered tree) называется корневое

Рисунок 5.7 Двоичные деревья. (а) Двоичное дерево, изображённое традиционным образом. Левый ребёнок вершины нарисован слева-снизу от неё, правый — справа-снизу. (б) Другое двоичное дерево (отличие в том, что теперь у вершины 7 есть правый ребёнок 5 и нет левого, а не наоборот). Такое различие возможно в двоичном дереве, но не в дереве с порядком на детях. (в) Добавляя дополнительные листья к дереву (а), мы получаем дерево, у которого каждая старая вершина имеет двух детей. Добавленные листья изображены квадратиками.

дерево с дополнительной структурой: для каждой вершины множество её детей упорядочено (известно, какой её потомок первый, какой второй и т. д.). Два дерева на рис. 5.6 одинаковы как корневые деревья, но различны как деревья с порядком на детях.

5.5.3 Двоичные деревья. Позиционные деревья

Двоичное дерево (binary tree) проще всего определить рекурсивно как конечный набор вершин, который

- либо пуст (не содержит вершин),
- либо разбит на три непересекающиеся части: вершину, называемую **корнем** (root), двоичное дерево, называемое **левым поддеревом** (left subtree) корня, и двоичное дерево, называемое **правым поддеревом** (right subtree) корня.

Двоичное дерево, не содержащее вершин, называется **пустым** (empty). Оно иногда обозначается NIL. Если левое поддерево непусто, то его корень называется **левым ребёнком** (left child) корня всего дерева; **правый ребёнок** (right child) определяется аналогично. Если левое или правое поддерево корня пусто, то говорят, что у корня нет левого или правого ребёнка (child is absent). Пример двоичного дерева показан на рис. 5.7 (а).

Было бы ошибкой определить двоичное дерево просто как дерево с порядком на детях, в котором степень каждой вершины не превосходит 2. Дело в том, что в двоичном дереве важно, каким является единственный ребёнок вершины степени 1 — левым или правым, а для дерева с порядком на детях такого различия не существует. На рис. 5.7 (а,б) показаны два различных двоичных дерева, которые высота=3, глубина 0, глубина 1 и т.п.

Рисунок 5.8 Полное двоичное дерево высоты 3 имеет 8 листьев и 7 внутренних вершин.

одинаковы как деревья с порядком на детях.

Пустующие места в двоичном дереве часто заполняют фиктивными листьями. После этого у каждой старой вершины будет двое детей (либо прежних, либо добавленных). Это преобразование показано на рис. 5.7 (в).

Можно определить аналоги двоичных деревьев для деревьев большей степени: двоичные деревья являются частным случаем **k-ичных** (k-ary) деревьев при $k = 2$. Более подробно, **позиционное дерево** (positional tree) определяется как корневое дерево, в котором дети любой вершины помечены различными целыми положительными числами, которые считаются их номерами. При этом у каждой вершины есть вакансии для детей номер 1, 2, 3 и так далее, из которых некоторые (конечное число) заполнены, а остальные свободны (ith child is absent). При этом k-ичным деревом называется позиционное дерево, не имеющее вершин с номерами больше k.

Полным k-ичным деревом (complete k-ary tree) называется k-ичное дерево, в котором все листья имеют одинаковую глубину и все внутренние вершины имеют степень k. (Тем самым структура такого дерева полностью определяется его высотой.) На рис. 5.8 показано полное двоичное дерево высоты 3. Подсчитаем, сколько листьев имеет полное k-ичное дерево высоты h. Корень является единственной вершиной глубины 0, его k детей являются вершинами глубины 1, их детьми являются k^2 вершин глубины k и так далее вплоть до k^h листьев глубины h. Можно добавить, что высота k-ичного дерева с n листьями равна $\log_k n$ (такое дерево существует, только если этот логарифм целый). Число внутренних вершин полного k-ичного дерева высоты h равно

$$1 + k + k^2 + \ldots + k^{h-1} = \frac{k^h - 1}{k - 1}$$

(см. (3.3)). В частности, для полного двоичного дерева число внутренних вершин на единицу меньше числа листьев.

Упражнения

5.5-1 Нарисуйте все деревья (без выделенного корня), содержащие три вершины A, B и C. Нарисуйте все корневые деревья с вершинами A, B и C и корнем A. Нарисуйте все (корневые) деревья с порядком на детях с вершинами A, B и C и корнем A. Нарисуйте все двоичные деревья с вершинами A, B и C и корнем A.

5.5-2 Покажите, что для любого $n \geqslant 7$ существует дерево с n вершинами, из которого можно получить n различных корневых деревьев, объявив корнем одну из n вершин.

5.5-3 Пусть $G = (V, E)$ — ориентированный ациклический граф, в котором существует вершина v_0, из которой в каждую другую $v \in V$ ведёт единственный путь. Покажите, что неориентированный вариант графа G является деревом.

5.5-4 Докажите по индукции, что в любом двоичном дереве число вершин степени 2 на единицу меньше числа листьев.

5.5-5 Покажите, что двоичное дерево с n вершинами имеет высоту не меньше $\lfloor \lg n \rfloor$.

5.5-6* Определим **внутреннюю сумму длин** (internal path length) для двоичного дерева, в котором каждая вершина имеет степень 0 или 2, как сумму глубин всех внутренних вершин. Определим **внешнюю сумму длин** для этого же дерева как сумму глубин всех его листьев. Пусть n — число внутренних вершин такого дерева, i и e — внутренняя и внешняя суммы длин. Докажите, что $e = i + 2n$.

5.5-7* Определим *"вес"* листа в двоичном дереве как 2^{-d}, где d — его глубина. Докажите, что сумма весов всех листьев в двоичном дереве не превосходит 1 (**неравенство Крафта**, Kraft inequality)

5.5-8* Покажите, что в любом двоичном дереве с L листьями можно найти поддерево, число листьев в котором находится на отрезке $[L/3, 2L/3]$.

Задачи

5-1 Раскраска графа

Назовём k-раскраской неориентированного графа (V, E) функцию $c\colon V \to \{0, 1, \ldots, k-1\}$, для которого $c(u) \neq c(v)$ для любых двух смежных вершин u и v. (Концы любого ребра должны иметь разные цвета.)

а. Покажите, что любое дерево имеет 2-раскраску.

б. Покажите, что следующие свойства неориентированного графа G равносильны:

1. Граф G двудольный.
2. Граф G имеет 2-раскраску.
3. Граф G не имеет циклов нечётной длины.

в. Пусть d — максимальная степень вершин неориентированного графа G. Покажите, что G имеет $(d+1)$-раскраску.

г. Покажите, что если граф G имеет $O(|V|)$ рёбер, то G имеет $O(\sqrt{|V|})$-раскраску.

5-2 Графы и люди

Переведите на язык неориентированных графов следующие утверждения и докажите их. (Предполагается, что отношение "*быть другом*" симметрично, и человек не включается в число своих друзей.)

а. В любой компании из $n \geqslant 2$ человек найдутся два человека с одинаковым числом друзей (среди присутствующих).

б. В любой группе из 6 человек можно найти либо трёх человек, являющихся друзьями друг друга, либо трёх человек, никакие двое из которых не являются друзьями.

в. Любую компанию людей можно развести по двум комнатам так, что для каждого человека как минимум половина его друзей окажутся в другой комнате.

г. Если в компании из n человек у каждого не менее $n/2$ друзей, то эту компанию можно рассадить за круглым столом так, чтобы каждый сидел между двумя своими друзьями.

5-3 Разбиение деревьев на части

Многие алгоритмы на графах, действующие по принципу "*разделяй и властвуй*", делят граф на две части, при этом желательно удалять как можно меньше рёбер и получить две части по возможности близкого размера.

а. Покажите, что в любом двоичном дереве с n вершинами можно найти ребро, после удаления которого получатся две части размера не больше $3n/4$ каждая.

б. Покажите, что константу $3/4$ в пункте (а) нельзя улучшить, приведя пример двоичного дерева, для которого после удаления любого ребра в одной из частей остаётся не менее $3n/4$ вершин.

в. Покажите, что множество вершин любого двоичного дерева с n вершинами можно разбить на две части A и B таким образом, что $|A| = \lfloor n/2 \rfloor$, $|B| = \lceil n/2 \rceil$, а число рёбер, соединяющих вершины из разных частей, есть $O(\lg n)$.

Замечания

Основатель символической логики Буль (G. Boole) ввёл многие из нынешних теоретико-множественных обозначений в книге, опубликованной в 1854 году. Современная теория множеств (прежде всего теория мощностей бесконечных множеств) была создана Кантором (G. Cantor) в 1874–1895 годах. Термин *"функция"* использовал Лейбниц (G.W. Leibnitz) в применении к некоторым математическим формулам. Определение функции впоследствии многократно обобщалось. Теория графов восходит к 1736 году, когда Эйлер (L. Euler) показал, что невозможно пройти по всем семи мостам города Кёнигсберга по одному разу и вернуться в исходную точку.

Полезным справочником по определениям и результатам теории графов является книга Харари [94].

6 Комбинаторика и вероятность

В этой главе излагаются начала комбинаторики и теории вероятностей. Если вы знакомы с ними, советуем просмотреть начало главы и внимательно прочесть последние разделы. Многие главы этой книги не используют теорию вероятностей, но кое-где она необходима.

Раздел 6.1 напоминает основные факты комбинаторики (в том числе формулы для перестановок и сочетаний). Раздел 6.2 содержит аксиомы вероятности и основные факты о распределениях вероятностей. В разделе 6.3 определяются понятия случайной величины, ее математического ожидания и дисперсии. Раздел 6.4 посвящен геометрическому и биномиальному распределениям. Исследование биномиального распределения продолжается в разделе 6.5 (оценка *"хвостов"*). В последнем разделе 6.6 применение теории вероятностей иллюстрируется на примере трёх задач: парадокса дня рождения, случайного распределения шаров по урнам и оценки длины выигрышных участков при бросании монеты.

6.1 Подсчёт количеств

Иногда можно найти число предметов определённого вида, не перечисляя их все. Например, легко найти число всех n-битовых строк или всех перестановок n объектов. В этом разделе мы рассмотрим основные методы такого подсчёта. Предполагается, что читатель знаком с понятиями теории множеств (см. разд. 5.1).

Правила суммы и произведения

Множество, количество элементов которого мы хотим подсчитать, часто может быть представлено в виде объединения непересекающихся множеств или декартова произведения множеств.

Правило суммы (rule of sum) гласит, что $|A \cup B| = |A| + |B|$ для непересекающихся конечных множеств A и B (частный случай формулы (5.3)). Если символ на номере машины должен быть либо латинской буквой, либо цифрой, то всего есть $26+10=36$ возможностей, так как букв 26, а цифр 10.

Правило произведения (rule of product) утверждает, что $|A \times B| = |A| \cdot |B|$ для конечных множеств A и B, см. (5.4). Например, имея 28 сортов мороженого и 4 вида сиропа, можно изготовить $28 \cdot 4 = 112$ вариантов мороженого с сиропом (не смешивая разные сорта мороженого и сиропа).

Строки

Строкой (string; по-русски говорят также о **словах**) называют конечную последовательность элементов некоторого конечного множества S (называемого **алфавитом**). Например, существует 8 двоичных (составленных из нулей и единиц) строк длины 3:

$$000, 001, 010, 011, 100, 101, 110, 111.$$

Иногда строку длины k называют **k-строкой** (k-string). **Подстрокой** (substring) s' строки s называется произвольная последовательность идущих подряд элементов строки s. Говоря о **k-подстроке** (k-substring), имеют в виду подстроку длины k. Так, 010 является 3-подстрокой строки 01101001 (она начинается с позиции 4), а 111 — нет.

Строка длины k из элементов множества S является элементом прямого произведения S^k, так что всего существует $|S|^k$ строк длины k. В частности, имеется 2^k двоичных строк длины k. Это можно объяснить ещё и так: первый элемент строки можно выбрать $|S|$ способами; для каждого из них есть $|S|$ вариантов продолжения, и так далее — всего k выборов, получается $|S| \times |S| \times \ldots \times |S|$ (k множителей) вариантов.

Перестановки

Перестановкой (permutation) конечного множества S называется упорядоченная последовательность всех его элементов, в которой каждый элемент встречается ровно один раз. Так, существует 6 перестановок множества $S = \{a, b, c\}$:

$$abc, acb, bac, bca, cab, cba.$$

Всего имеется $n!$ перестановок множества из n элементов, так как первый элемент перестановки можно выбрать n способами, второй $n-1$ способами, третий $n-2$ способами, и т.д.

Размещения без повторений

Если каждый элемент можно использовать только один раз, но не требуется использовать все элементы, говорят о **размещениях без повторений**. Пусть фиксировано множество S из n элементов и некоторое k, не превосходящее n. Размещением без повторений из n по k называют последовательность длины k, составленную из различных элементов S. (Английский термин — k-permutation.) Число таких размещений равно

$$n(n-1)(n-2)\cdots(n-k+1) = \frac{n!}{(n-k)!} \tag{6.1}$$

так как существует n способов выбора первого элемента, $n-1$ способов выбора второго элемента, и так далее до k-го элемента, который можно выбрать $n-k+1$ способами.

Например, существует $12 = 4\cdot 3$ последовательностей из двух различных элементов множества $\{a, b, c, d\}$:

$$ab, ac, ad, ba, bc, bd, ca, cb, cd, da, db, dc.$$

Частным случаем этой формулы является формула для числа перестановок (поскольку перестановки являются частным случаем размещений при $n = k$).

Сочетания

Сочетаниями (k-combinations) из n элементов по k называются k-элементные подмножества какого-либо n-элементного множества. Например, у множества $\{a, b, c, d\}$ из 4 элементов имеется 6 двухэлементных подмножеств

$$\{a, b\}, \{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}, \{c, d\}.$$

Число сочетаний из n по k в $k!$ раз меньше числа размещений без повторений (для тех же n и k), так как из каждого k-сочетания можно сделать $k!$ размещений без повторений, переставляя его элементы. Поэтому из формулы (6.1) следует, что число сочетаний из n по k равно

$$\frac{n!}{k!(n-k)!}. \tag{6.2}$$

Для $k = 0$ эта формула даёт 1, как и должно быть (есть ровно одно пустое подмножество; напомним, что $0! = 1$).

Биномиальные коэффициенты

Для числа сочетаний из n по k используется обозначение C_n^k или (в английской литературе) $\binom{n}{k}$:

$$C_n^k = \binom{n}{k} = \frac{n!}{k!(n-k)!}. \qquad (6.3)$$

Эта формула симметрична относительно замены k на $n-k$:

$$C_n^k = C_n^{n-k}. \qquad (6.4)$$

Числа C_n^k известны также как **биномиальные коэффициенты** (binomial coefficients), появляющиеся в **биноме Ньютона** (binomial expansion):

$$(x+y)^n = \sum_{k=0}^{n} C_n^k x^k y^{n-k} \qquad (6.5)$$

(если раскрыть скобки в $(x+y)^n$, то количество членов, содержащих k множителей x и $n-k$ множителей y, равно количеству способов выбрать k мест из n, т.е. C_n^k).

При $x = y = 1$ бином Ньютона даёт

$$2^n = \sum_{k=0}^{n} C_n^k. \qquad (6.6)$$

(Комбинаторный смысл: 2^n двоичных строк длины n сгруппированы по числу единиц: имеется как раз C_n^k строк с k единицами.)

Существует много тождеств с биномиальными коэффициентами (некоторые из них предлагаются в качестве упражнений в конце этого раздела).

Оценки биномиальных коэффициентов

В некоторых случаях нам понадобится оценить величину биномиальных коэффициентов. Для $1 \leqslant k \leqslant n$ имеем оценку снизу

$$\begin{aligned} C_n^k &= \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1} \\ &= \left(\frac{n}{k}\right)\left(\frac{n-1}{k-1}\right)\cdots\left(\frac{n-k+1}{1}\right) \\ &\geqslant \left(\frac{n}{k}\right)^k. \end{aligned} \qquad (6.7)$$

Используя неравенство $k! \geqslant (\frac{k}{e})^k$, являющееся следствием формулы Стирлинга (2.12), получаем оценку сверху

$$\begin{aligned} C_n^k &= \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1} \\ &\leqslant \frac{n^k}{k!} && (6.8) \\ &\leqslant \left(\frac{en}{k}\right)^k . && (6.9) \end{aligned}$$

Для всех $0 \leqslant k \leqslant n$ можно по индукции (см. упр. 6.1-12) доказать оценку

$$C_n^k \leqslant \frac{n^n}{k^k(n-k)^{n-k}}, \qquad (6.10)$$

где ради удобства записи полагаем $0^0 = 1$. Для $k = \lambda n$, где $0 \leqslant \lambda \leqslant 1$, эта оценка может быть записана как

$$\begin{aligned} C_n^{\lambda n} &\leqslant \frac{n^n}{(\lambda n)^{\lambda n}((1-\lambda)n)^{(1-\lambda)n}} \\ &= \left(\left(\frac{1}{\lambda}\right)^{\lambda}\left(\frac{1}{1-\lambda}\right)^{1-\lambda}\right)^n && (6.11) \\ &= 2^{nH(\lambda)}, && (6.12) \end{aligned}$$

где величина

$$H(\lambda) = -\lambda \lg \lambda - (1-\lambda)\lg(1-\lambda) \qquad (6.13)$$

называется **(двоичной) шенноновской энтропией**, по-английски (binary) entropy function. В этой записи мы полагаем $0 \lg 0 = 0$, так что $H(0) = H(1) = 0$.

Упражнения

6.1-1 Сколько существует k-подстрок строки длины n? (Подстроки, начинающиеся с различных позиций строки, считаются разными.) каково общее число подстрок строки длины n?

6.1-2 **Булева функция** (boolean function) с n входами и m выходами — это функция, определенная на множестве $\{\text{TRUE}, \text{FALSE}\}^n$ со значениями в множестве $\{\text{TRUE}, \text{FALSE}\}^m$. Сколько существует различных булевых функций с n входами и одним выходом? А булевых функций с n входами и m выходами?

6.1-3 Сколькими способами n (различных) профессоров могут расположиться за круглым столом? Способы, отличающиеся поворотом, считаются одинаковыми.

6.1-4 Сколькими способами можно выбрать три различных числа из множества $\{1, 2, \ldots, 100\}$ так, чтобы их сумма была четной? (Порядок выбора существен.)

6.1-5 Докажите тождество

$$C_n^k = \frac{n}{k} C_{n-1}^{k-1} \tag{6.14}$$

для $0 < k \leqslant n$.

6.1-6 Докажите тождество

$$C_n^k = \frac{n}{n-k} C_{n-1}^{k}$$

для $0 \leqslant k < n$.

6.1-7 Выбирая k предметов из n, можно отметить один из предметов и следить, выбран он или нет. Используя это обстоятельство, докажите, что

$$C_n^k = C_{n-1}^k + C_{n-1}^{k-1}.$$

6.1-8 Используя результат упражнения 6.1-7, составьте таблицу для C_n^k при $n = 0, 1, 2, \ldots, 6$ и при k от 0 до n в виде равнобедренного треугольника (C_0^0 сверху, C_1^0 и C_1^1 в следующей строке, и так далее). Этот треугольник называют **треугольником Паскаля** (Pascal's triangle).

6.1-9 Докажите равенство

$$\sum_{i=1}^{n} i = C_{n+1}^2.$$

6.1-10 Покажите, что для фиксированного $n \geqslant 0$ величина C_n^k достигает наибольшего (среди всех k от 0 до n) значения при $k = \lfloor n/2 \rfloor$ и при $k = \lceil n/2 \rceil$ (так что для чётного n максимум один, а для нечётного — два стоящих рядом).

6.1-11* Покажите, что для любых $n \geqslant 0$, $j \geqslant 0$, $k \geqslant 0$, $j + k \leqslant n$ выполнено неравенство

$$C_n^{j+k} \leqslant C_n^j C_{n-j}^k$$

с помощью комбинаторных рассуждений, а также с использованием формулы (6.3). В каких случаях это неравенство обращается в равенство?

6.1-12* Докажите по индукции неравенство (6.10) для $k \leqslant n/2$; затем, используя (6.4), докажите его для всех $k \leqslant n$.

6.1-13* Используя формулу Стирлинга, докажите, что

$$C_{2n}^{n} = \frac{2^{2n}}{\sqrt{\pi n}}(1 + O(1/n)). \qquad (6.16)$$

6.1-14* Дифференцируя $H(\lambda)$, покажите, что максимум достигается при $\lambda = 1/2$. Чему равно $H(1/2)$?

6.2 Вероятность

В этом разделе мы напомним основные понятия теории вероятностей.

Пусть задано некоторое множество S, которое мы называем **вероятностным пространством** (sample space), а его элементы — **элементарными событиями** (elementary events). Каждый из элементов может рассматриваться как возможный исход испытания. Например, бросанию двух различных монет соответствует вероятностное пространство, содержащее четыре строки длины 2, составленные из символов О (орёл) и Р (решка):

$$S = \{\text{ОО}, \text{ОР}, \text{РО}, \text{РР}\}$$

Событием (event) называется подмножество пространства S. Например, в нашем примере можно рассмотреть событие *"выпал один орёл и одна решка"*, т.е. множество $\{\text{ОР}, \text{РО}\}$.

Событие S (всё вероятностное пространство) называется **достоверным событием** (certain event), а событие $\varnothing$ называется **невозможным событием** (null event). События A и B называются **несовместными** (mutually exclusive), если $A \cap B = \varnothing$. Каждое элементарное событие $s \in S$ мы будем считать событием $\{s\} \subseteq S$. Различные элементарные события несовместны.

Сказанное относится без оговорок к случаю конечного или счётного множества S. В общем случае определение сложнее, и событиями считаются не все подмножества множества S, а только некоторые. Они должны образовывать σ-алгебру (пересечение и объединение счётного числа событий есть событие; дополнение события есть событие). Мы не будем говорить об этом подробно, хотя некоторые примеры такого рода (равномерное распределение на отрезке) нам встретятся.

Аксиомы вероятности

Распределением вероятностей (probability distribution) на вероятностном пространстве S называется функция P, ставящая в соответствие каждому событию некоторое неотрицательное число и удовлетворяющая следующим требованиям (**аксиомам вероятности**, по-английски probability axioms):

1. $\mathsf{P}A \geqslant 0$ для любого события A.
2. $\mathsf{P}\{S\} = 1$.
3. $\mathsf{P}\{A \cup B\} = \mathsf{P}\{A\} + \mathsf{P}\{B\}$ для любых двух несовместных событий A и B, и, более того,

$$\mathsf{P}\{\cup_i A_i\} = \sum_i \mathsf{P}\{A_i\}.$$

для любой (конечной или счетной) последовательности попарно несовместных событий $A_1, A_2, \ldots$

Число $\mathsf{P}\{A\}$ называется **вероятностью события** A (probability of the event A). Заметим, что аксиома 2 фиксирует *"единицу измерения"* вероятностей, принимая за 1 вероятность достоверного события.

Вот несколько простых следствий из этих аксиом. Невозможное событие имеет нулевую вероятность $\mathsf{P}\{\varnothing\} = 0$. Если $A \subseteq B$, то $\mathsf{P}\{A\} \leqslant \mathsf{P}\{B\}$. Используя обозначение $\overline{A}$ для события $S - A$ (дополнение к A), имеем $\mathsf{P}\{\overline{A}\} = 1 - \mathsf{P}\{A\}$. Для любых двух событий A и B имеет место

$$\begin{aligned} \mathsf{P}\{A \cup B\} &= \mathsf{P}\{A\} + \mathsf{P}\{B\} - \mathsf{P}\{A \cap B\} && (6.17) \\ &\leqslant \mathsf{P}\{A\} + \mathsf{P}\{B\}. && (6.18) \end{aligned}$$

В нашем примере с бросанием двух монет, положим вероятность каждого элементарного исхода равной 1/4. Тогда вероятность выпадения по крайней мере одного орла будет

$$\begin{aligned} \mathsf{P}\{\text{оо}, \text{ор}, \text{ро}\} &= \mathsf{P}\{\text{оо}\} + \mathsf{P}\{\text{ор}\} + \mathsf{P}\{\text{ро}\} \\ &= 3/4. \end{aligned}$$

Иначе: вероятность того, что не будет ни одного орла, равна $\mathsf{P}\{\text{рр}\} = 1/4$, поэтому вероятность появления по меньшей мере одного орла есть $1 - 1/4 = 3/4$.

6.2.1 Дискретное распределение вероятностей

Распределение вероятностей на конечном или счётном вероятностном пространстве называется **дискретным** (discrete). Для таких распределений можно написать

$$\mathsf{P}\{A\} = \sum_{s \in A} \mathsf{P}\{s\},$$

для любого события A, поскольку оно является объединением не более чем счётного множества несовместных элементарных событий. Если множество S конечно и все элементы его равновероятны, то получается **равномерное распределение вероятностей** (uniform probability distribution) на конечном множестве S. При этом вероятность любого события, включающего в себя k элементарных исходов из $|S|$, равна $k/|S|$. В таких случаях говорят *"выберем случайно элемент $s \in S$"*.

В качестве примера рассмотрим **бросание симметричной монеты** (flipping a fair coin), для которой вероятности орла и решки одинаковы и равны $1/2$. Бросая её n раз, мы приходим к равномерному распределению на пространстве $S = \{\text{о}, \text{р}\}^n$, состоящем из 2^n элементов. Каждое элементарное событие из S можно рассматривать как строку длины n элементов множества $\{\text{о}, \text{р}\}$, и все такие строки имеют вероятность $1/2^n$. Событие

$$A = \{\text{выпало } k \text{ орлов и } n - k \text{ решек}\}$$

есть подмножество S и состоит из $|A| = C_n^k$ элементов, так как существует C_n^k строк, содержащих ровно k орлов. Тем самым, вероятность события A равна $\mathsf{P}\{A\} = C_n^k/2^n$.

Непрерывное равномерное распределение вероятностей

Будем считать элементарными исходами точки некоторого отрезка $[a, b]$. Определим вероятность события $[c, d] \subseteq [a, b]$ формулой

$$\mathsf{P}\{[c, d]\} = \frac{d - c}{b - a}.$$

В этом случае, как мы говорили, надо считать событиями не все подмножества отрезка, а только некоторые, которые называют измеримыми. Мы не приводим соответствующих определений, отсылая читателя к любому учебнику по теории вероятностей или по теории меры.

Заметим, что вероятность каждой точки равна 0, и потому вероятность полуинтервала $(c, d]$ и интервала (c, d) могут быть определены той же формулой.

Такое распределение вероятностей называют **непрерывным равномерным распределением** (continuous uniform probability distribution).

Условная вероятность и независимость

Иногда мы располагаем частичной информацией о результате эксперимента. Например, пусть нам известно, что в результате бросания двух симметричных монет по крайней мере одна из них выпала орлом. Какова вероятность того, что обе монеты выпали орлом? Известная нам информация позволяет исключить случай выпадения двух решек. Три оставшихся исхода будут равновероятны, поэтому вероятность каждого (в том числе и интересующего нас) есть $1/3$.

Эта идея формализуется в определении **условной вероятности** (conditional probability) события A при условии события B; она обозначается $\mathsf{P}\{A|B\}$ и определяется формулой

$$\mathsf{P}\{A|B\} = \frac{\mathsf{P}\{A \cap B\}}{\mathsf{P}\{B\}}, \qquad (6.19)$$

(мы предполагаем, что $\mathsf{P}\{B\} \neq 0$). Интуитивный смысл понятен: событие B происходит в некоторой доле экспериментов; мы смотрим, какую часть среди них составляют те, когда произошло ещё и событие A.

Два события называются **независимыми** (independent), если

$$\mathsf{P}\{A \cap B\} = \mathsf{P}\{A\}\mathsf{P}\{B\},$$

В случае $\mathsf{P}\{B\} \neq 0$ это условие можно переписать как

$$\mathsf{P}\{A|B\} = \mathsf{P}\{A\}.$$

В нашем примере с двукратным бросанием монеты появления орла при первом и втором бросании будут независимыми, так как каждое событие имеет вероятность 1/2, а их пересечение (два орла) — 1/4. В том же примере события *"первая монета выпала орлом"* и *"выпал один орёл и одна решка"* также независимы, хотя это сразу и не так ясно. Но в этом легко убедиться по определению: вероятность каждого события равна 1/2, вероятность их пересечения равна 1/4. А вот события *"первая монета выпала орлом"* и *"выпала хоть одна решка"* не будут независимыми.

События *"первая монета выпала орлом"* и *"вторая монета выпала орлом"* перестанут быть независимыми, если изменить распределение вероятностей и считать, что монеты склеены и одновременно выпадают либо орлом, либо решкой (т. е. что комбинации ОО и РР имеют вероятность 1/2).

События $A_1, A_2, \ldots, A_n$ называются **попарно независимыми** (pairwise independent), если

$$\mathsf{P}\{A_i \cap A_j\} = \mathsf{P}\{A_i\}\mathsf{P}\{A_j\}$$

для всех $1 \leqslant i < j \leqslant n$.

События $A_1, A_2, \cdots, A_n$ называются **независимыми в совокупности** (mutually independent), если для любого набора $A_{i_1}, A_{i_2}, \ldots, A_{i_k}$ этих событий (здесь $2 \leqslant k \leqslant n$ и $1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant n$) имеет место равенство

$$\mathsf{P}\{A_{i_1} \cap A_{i_2} \cap \ldots \cap A_{i_k}\} = \mathsf{P}\{A_{i_1}\}\mathsf{P}\{A_{i_2}\} \cdots \mathsf{P}\{A_{i_k}\}.$$

Это требование — более сильное: например, в нашем примере события *"первая монета выпала орлом"*, *"вторая монета выпала орлом"* и *"две монеты выпали одинаково"* попарно независимы, но не являются независимыми в совокупности.

Формула Байеса

Из определения условной вероятности (6.19) следует, что для двух событий A и B, вероятности которых положительны, выполнено равенство

$$\begin{aligned} \mathrm{P}\{A \cap B\} &= \mathrm{P}\{B\}\mathrm{P}\{A|B\} \\ &= \mathrm{P}\{A\}\mathrm{P}\{B|A\}. \end{aligned} \tag{6.20}$$

Выражая отсюда $\mathrm{P}\{A|B\}$, получаем формулу

$$\mathrm{P}\{A|B\} = \frac{\mathrm{P}\{A\}\mathrm{P}\{B|A\}}{\mathrm{P}\{B\}}, \tag{6.21}$$

известную как **формула Байеса** (Bayes's theorem). Эту формулу можно переписать так: поскольку $B = (B \cap A) \cup (B \cap \overline{A})$, а $B \cap A$ и $B \cap \overline{A}$ — несовместные события, то

$$\begin{aligned} \mathrm{P}\{B\} &= \mathrm{P}\{B \cap A\} + \mathrm{P}\{B \cap \overline{A}\} \\ &= \mathrm{P}\{A\}\mathrm{P}\{B|A\} + \mathrm{P}\{\overline{A}\}\mathrm{P}\{B|\overline{A}\}. \end{aligned}$$

Подставляя данное выражение в формулу (6.21), получаем другой вариант формулы Байеса:

$$\mathrm{P}\{A|B\} = \frac{\mathrm{P}\{A\}\mathrm{P}\{B|A\}}{\mathrm{P}\{A\}\mathrm{P}\{B|A\} + \mathrm{P}\{\overline{A}\}\mathrm{P}\{B|\overline{A}\}}.$$

Формула Байеса помогает вычислять условные вероятности. Пусть у нас есть две монеты: одна симметричная, а другая всегда выпадает орлом. Мы случайным образом выбираем одну из двух монет, после чего её дважды подбрасываем. Предположим, что оба раза выпали орлы. Какова вероятность того, что была выбрана несимметричная монета?

Решим эту задачу при помощи формулы Байеса. Пусть событие A — выбор несимметричной монеты, событие B — выпадение выбранной монеты орлами дважды. Нам нужно вычислить $\mathrm{P}\{A|B\}$. Имеем: $\mathrm{P}\{A\} = 1/2$, $\mathrm{P}\{B|A\} = 1$, $\mathrm{P}\{\overline{A}\} = 1/2$ и $\mathrm{P}\{B|\overline{A}\} = 1/4$, следовательно,

$$\mathrm{P}\{B|A\} = \frac{(1/2) \cdot 1}{(1/2) \cdot 1 + (1/2) \cdot (1/4)} = 4/5.$$

Упражнения

6.2-1 Докажите **неравенство Буля** (Boole's inequality):

$$\mathrm{P}\{A_1 \cup A_2 \cup \ldots\} \leqslant \mathrm{P}\{A_1\} + \mathrm{P}\{A_2\} + \ldots \tag{6.22}$$

для любой конечной или счётной последовательности событий $A_1, A_2, \ldots$.

6.2-2 Профессор бросает симметричную монету, а студент бросает бросает две симметричные монеты. Какова вероятность того, что у профессора выпадет больше орлов, чем у студента? (Все три бросания независимы.)

6.2-3 Колоду карт (с числами от 1 до 10) тасуют и вынимают три карты. Какова вероятность того, что числа на этих картах будут идти в возрастающем порядке?

6.2-4* Имеется несимметричная монета, для которой вероятность выпадения орла есть неизвестное нам число p $(0 < p < 1)$. Покажите, как с её помощью можно имитировать симметричную монету, сделав несколько бросаний. (Указание: бросьте монету дважды; если результаты разные, дайте ответ; если одинаковые, повторяйте испытание.)

6.2-5* Как имитировать бросание монеты с вероятностью появления орла a/b, имея симметричную монету, которую можно подбрасывать несколько раз? (Числа a и b целые, $0 < a < b$, математическое ожидание числа бросаний должно быть ограничено сверху полиномом от $\lg b$.)

6.2-6 Докажите, что

$$\mathsf{P}\{A|B\} + \mathsf{P}\{\overline{A}|B\} = 1.$$

6.2-7 Докажите, что для любого набора событий $A_1, A_2, \ldots, A_n$,

$$\begin{aligned}&\mathsf{P}\{A_1 \cap A_2 \cap \ldots \cap A_n\} =\\ &= \mathsf{P}\{A_1\} \cdot \mathsf{P}\{A_2|A_1\} \cdot \mathsf{P}\{A_3|A_1 \cap A_2\} \cdots \mathsf{P}\{A_n|A_1 \cap A_2 \cap \cdots \cap A_{n-1}\}.\end{aligned}$$

6.2-8* Придумайте множество из n попарно независимых событий, для которого любое подмножество из $k > 2$ событий не будет независимым в совокупности.

6.2-9* События A и B являются **условно независимыми** (conditionally independent) при условии события C, если

$$\mathsf{P}\{A \cap B|C\} = \mathsf{P}\{A|C\} \cdot \mathsf{P}\{B|C\}.$$

Придумайте простой (но не тривиальный) пример двух событий, которые не являются независимыми, но условно независимы при условии некоторого третьего события.

6.2-10* Вы участвуете в игре, в которой приз скрыт за одной из трёх ширм (и выиграете приз, если отгадаете, где). После того как вы выбрали одну из ширм, ведущий игры открыл одну из двух оставшихся ширм, и оказалось, что там приза нет. В этот момент можно поменять свой выбор, указав на третью ширму. Как изменятся шансы на выигрыш, если вы сделаете это?

6.2-11* Начальник тюрьмы выбрал одного из трёх заключённых X, Y и Z, чтобы отпустить его на волю. Остальные двое будут казнены. Страж знает, кто из троих выйдет на свободу, но не имеет права сообщать никакому из узников информацию о его судьбе. Заключенный X просит стража назвать ему имя одного из заключенных Y или Z, который будет казнён, объясняя, что ему и так известно, что один из них точно будет казнён, а, значит, он не получит никакой информации о своей судьбе. Страж сообщает X, что Y будет казнен. Заключённый X радуется, считая, что его шансы остаться в живых возросли до 1/2 (освобождён будет или он, или Z). Прав ли он?

6.3 Дискретные случайные величины

Дискретная случайная величина (discrete random variable) X — это функция, отображающая конечное или счётное вероятностное пространство S в множество действительных чисел. Каждому возможному исходу испытания она ставит в соответствие действительное число. (Тем самым на множестве значений функции X возникает распределение вероятностей.)

В теории вероятностей рассматривают и случайные величины на несчётных вероятностных пространствах, но это сложнее, и мы обойдёмся без них.

Для случайной величины X и действительного числа x определяем событие $X = x$ как $\{s \in S : X(s) = x\}$; вероятность этого события равна

$$\mathsf{P}\{X = x\} = \sum_{\{s \in S : X(s) = x\}} \mathsf{P}\{s\}.$$

Функция

$$f(x) = \mathsf{P}\{X = x\}$$

называется **функцией распределения вероятностей** (probability density function) случайной величины X. Из аксиом вероятности следует, что $\mathsf{P}\{X = x\} \geqslant 0$ и $\sum_x \mathsf{P}\{X = x\} = 1$.

Для примера рассмотрим бросание пары обычных шестигранных костей. Имеется 36 элементарных событий, составляющих вероятностное пространство. Будем предполагать, что все они равновероятны: $\mathsf{P}\{s\} = 1/36$. Определим случайную величину X, как *максимальное число*, выпавшее на одной из двух костей. Тогда

$\mathsf{P}\{X = 3\} = 5/36$, так как X принимает значение 3 при 5 элементарных исходах (а именно, $(1,3)$, $(2,3)$, $(3,3)$, $(3,2)$ и $(3,1)$).

Как правило, на одном и том же вероятностном пространстве рассматривают несколько случайных величин. Если X и Y — случайные величины, то функция

$$f(x,y) = \mathsf{P}\{X = x, Y = y\}$$

называется **функцией совместного распределения вероятностей** (joint probability density function) величин X и Y. Для фиксированного значения y

$$\mathsf{P}\{Y = y\} = \sum_x \mathsf{P}\{X = x,\ Y = y\}.$$

Аналогично, для фиксированного значения x,

$$\mathsf{P}\{X = x\} = \sum_y \mathsf{P}\{X = x,\ Y = y\}.$$

Используя определение условной вероятности (6.19), можно записать

$$\mathsf{P}\{X = x | Y = y\} = \frac{\mathsf{P}\{X = x,\ Y = y\}}{\mathsf{P}\{Y = y\}}.$$

Две случайные величины называются **независимыми** (independent), если события $X = x$ и $Y = y$ являются независимыми для любых значений x и y, другими словами, если если $\mathsf{P}\{X = x,\ Y = y\} = \mathsf{P}\{X = x\}\mathsf{P}\{Y = y\}$ для всех x и y.

Складывая и умножая случайные величины, определённые на одном и том же вероятностном пространстве, мы получаем новые случайные величины, определённые на том же пространстве.

Математическое ожидание случайной величины

Простейшая и наиболее часто используемая характеристика случайной величины — это её **среднее** (mean), называемое также **математическим ожиданием** (expected value, expectation). Для дискретной случайной величины X оно определяется формулой

$$\mathsf{M}[X] = \sum_x x\mathsf{P}\{X = x\}, \tag{6.23}$$

и существует, когда этот ряд имеет конечное число членов или абсолютно сходится. Иногда математическое ожидание обозначается μ_X или просто μ, если из контекста ясно, о какой случайной величине идет речь.

Пусть в игре дважды бросают симметричную монету; вы получаете 3 рубля за каждого выпавшего орла и отдаёте 2 рубля за каждую выпавшую решку. Выигрыш X будет случайной величиной, и её математическое ожидание будет равно

$$\begin{aligned}\mathsf{M}[X] &= 6 \cdot \mathsf{P}\{2\ \text{орла}\} + 1 \cdot \mathsf{P}\{1\ \text{орёл и}\ 1\ \text{решка}\} - 4 \cdot \mathsf{P}\{2\ \text{решки}\} \\ &= 6(1/4) + 1(1/2) - 4(1/4) \\ &= 1.\end{aligned}$$

Математическое ожидание суммы случайных величин равно сумме их ожиданий:

$$\mathsf{M}[X+Y] = \mathsf{M}[X] + \mathsf{M}[Y], \tag{6.24}$$

если $\mathsf{M}[X]$ и $\mathsf{M}[Y]$ определены. Это правило можно распространить на любые конечные и абсолютно сходящиеся бесконечные суммы.

Пусть X — случайная величина, а $g(x)$ — произвольная функция. Тогда можно рассмотреть случайную величину $g(X)$ (на том же вероятностном пространстве). Её математическое ожидание (если оно определено) можно найти по формуле

$$\mathsf{M}[g(X)] = \sum_x g(x)\mathsf{P}\{X = x\}.$$

Для функции $g(x) = ax$, где a — некоторая константа, имеем

$$\mathsf{M}[aX] = a\mathsf{M}[X]. \tag{6.25}$$

Два последних свойства можно скомбинировать в одной формуле (свойство линейности): для любых двух случайных величин X и Y и любой константы a

$$\mathsf{M}[aX+Y] = a\mathsf{M}[X] + \mathsf{M}[Y]. \tag{6.26}$$

Если две случайные величины X и Y независимы и их математические ожидания определены, то

$$\begin{aligned}\mathsf{M}[XY] &= \sum_x \sum_y xy\mathsf{P}\{X = x, Y = y\} \\ &= \sum_x \sum_y xy\mathsf{P}\{X = x\}\mathsf{P}\{Y = y\} \\ &= \Big(\sum_x x\mathsf{P}\{X = x\}\Big)\Big(\sum_y y\mathsf{P}\{Y = y\}\Big) \\ &= \mathsf{M}[X]\mathsf{M}[Y].\end{aligned}$$

Более общо, если имеется n независимых в совокупности случайных величин $X_1, X_2, \ldots, X_n$, имеющих математические ожидания, то

$$\mathsf{M}[X_1X_2\cdots X_n] = \mathsf{M}[X_1]\mathsf{M}[X_2]\cdots\mathsf{M}[X_n]. \tag{6.27}$$

Если случайная величина X может принимать только натуральные значения $(0, 1, 2, \ldots)$, то имеется красивая формула для её математического ожидания:

$$\begin{aligned} \mathsf{M}[X] &= \sum_{i=0}^{\infty} i\mathsf{P}\{X = i\} \\ &= \sum_{i=0}^{\infty} i(\mathsf{P}\{X \geqslant i\} - \mathsf{P}\{X \geqslant i+1\}) \\ &= \sum_{i=1}^{\infty} \mathsf{P}\{X \geqslant i\}. \end{aligned} \tag{6.28}$$

В самом деле, каждый член $\mathsf{P}\{X \geqslant i\}$ присутствует в сумме i раз со знаком плюс и $i-1$ раз со знаком минус (исключение составляет член $\mathsf{P}\{X \geqslant 0\}$, вовсе отсутствующий в сумме).

Дисперсия и стандартное отклонение

Дисперсия (variance) случайной величины X с математическим ожиданием $\mathsf{M}[X]$ определяется как

$$\begin{aligned} \mathsf{D}[X] &= \mathsf{M}[(X - \mathsf{M}[X])^2] \\ &= \mathsf{M}[X^2 - 2X\mathsf{M}[X] + \mathsf{M}^2[X]] \\ &= \mathsf{M}[X^2] - 2\mathsf{M}[X\mathsf{M}[X]] + \mathsf{M}^2[X] \\ &= \mathsf{M}[X^2] - 2\mathsf{M}^2[X] + \mathsf{M}^2[X] \\ &= \mathsf{M}[X^2] - \mathsf{M}^2[X]. \end{aligned} \tag{6.29}$$

Переходы $\mathsf{M}[\mathsf{M}^2[X]] = \mathsf{M}^2[X]$ и $\mathsf{M}[X\mathsf{M}[X]] = \mathsf{M}^2[X]$ законны, так как $\mathsf{M}[X]$ — это число (а не случайная величина) и можно сослаться на (6.25), полагая $a = \mathsf{M}[X]$. Формулу (6.29) можно переписать так:

$$\mathsf{M}[X^2] = \mathsf{D}[X] + \mathsf{M}^2[X] \tag{6.30}$$

При увеличении случайной величины в a раз её дисперсия растёт в a^2 раз:

$$\mathsf{D}[aX] = a^2\mathsf{D}[X].$$

Если X и Y независимы, то

$$\mathsf{D}[X + Y] = \mathsf{D}[X] + \mathsf{D}[Y].$$

Более общо, дисперсия суммы n попарно независимых случайных величин $X_1, \ldots, X_n$ равна сумме их дисперсий:

$$\mathsf{D}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathsf{D}[X_i]. \tag{6.31}$$

Стандартным отклонением (standard deviation) случайной величины X называется квадратный корень из её дисперсии. Часто стандартное отклонение случайной величины обозначается σ_X или просто σ, если из контекста ясно, о какой случайной величине идет речь. В этой записи дисперсия обозначается σ^2.

Упражнения

6.3-1 Подбрасываются две обычные шестигранные кости. Чему равно математическое ожидание суммы выпавших чисел? Чему равно математическое ожидание максимума из двух выпавших чисел?

6.3-2 В массиве $A[1\,..\,n]$ имеется n расположенных в случайном порядке различных чисел; все возможные расположения чисел равновероятны. Чему равно математическое ожидание номера места, на котором находится максимальный элемент? Номера места, на котором находится минимальный элемент?

6.3-3 В коробочке лежат три игральные кости. Игрок ставит доллар на одно из чисел от 1 до 6. Коробочка встряхивается и открывается. Если названное игроком число не выпало вовсе, то он проигрывает свой доллар. В противном случае он сохраняет его и получает дополнительно столько долларов, сколько выпало костей с названным им числом. Сколько в среднем выигрывает игрок в одной партии?

6.3-4* Пусть X и Y — независимые случайные величины. Покажите, что $f(X)$ и $g(Y)$ также независимы для любых функций f и g.

6.3-5* Пусть X — неотрицательная случайная величина с математическим ожиданием $\mathsf{M}[X]$. Докажите **неравенство Маркова** (Markov's inequality)

$$\mathsf{P}\{X \geqslant t\} \leqslant \mathsf{M}[X]/t \tag{6.32}$$

для всех $t > 0$. [Это неравенство называют также **неравенством Чебышёва**.]

6.3-6* Пусть S — вероятностное пространство, на котором определены случайные величины X и X', причём $X(s) \geqslant X'(s)$ для всех $s \in S$. Докажите, что для любого действительного числа t,

$$\mathsf{P}\{X \geqslant t\} \geqslant \mathsf{P}\{X' \geqslant t\}.$$

6.3-7 Что больше: математическое ожидание квадрата случайной величины или квадрат её математического ожидания?

6.3-8 Покажите, что для случайной величины, принимающей только значения 0 и 1, выполнено равенство $D[X] = \mathsf{M}[X]\mathsf{M}[1-X]$.

6.3-9 Выведите из определения дисперсии (6.29), что $\mathsf{D}[aX] = a^2\mathsf{D}[X]$.

6.4 Геометрическое и биномиальное распределения

Бросание симметричной монеты — частный случай **испытаний по схеме Бернулли** (Bernouilli trials) в которой рассматривается n независимых в совокупности испытаний, каждое из которых имеет два возможных исхода: **успех** (success), происходящий с вероятностью p, и **неудачу** (failure), имеющую вероятность $1-p$. Два важных распределения вероятностей — геометрическое и биномиальное — связаны со схемой Бернулли.

Геометрическое распределение

Рассмотрим серию испытаний Бернулли, в каждом из которых успех имеет вероятность p (а неудача имеет вероятность $q = 1 - p$). Какое испытание будет первым успешным? Пусть случайная величина X — его номер; эта величина принимает значения $1, 2, \ldots$, причём

$$\mathsf{P}\{X = k\} = q^{k-1}p \tag{6.33}$$

(первый успех будет иметь номер k, если $k-1$ испытаний до него были неудачными, а k-е оказалось удачным). Распределение вероятностей, заданное формулой (6.33), называется **геометрическим распределением** (geometric distribution). Оно показано на рис. 6.1.

Предполагая, что $p < 1$, найдём математическое ожидание геометрического распределения, используя формулу (3.6):

$$\mathsf{M}[X] = \sum_{k=1}^{\infty} kq^{k-1}p = \frac{p}{q}\sum_{k=0}^{\infty} kq^k = \frac{p}{q}\cdot\frac{q}{(1-q)^2} = 1/p. \tag{6.34}$$

Другими словами, нужно в среднем $1/p$ раз провести испытание, чтобы добиться успеха, что естественно ожидать, поскольку вероятность успеха равна p. Дисперсию можно вычислить аналогичным образом; получится, что

$$\mathsf{D}[X] = q/p^2. \tag{6.35}$$

Пример: будем бросать пару костей, пока в сумме не выпадет или семь или одиннадцать. Для одного эксперимента есть 36 возможных

Рисунок 6.1 Геометрическое распределение с вероятностью успеха $p = 1/3$ и вероятностью неудачи $q = 1 - p$. Математическое ожидание равно $1/p = 3$.

исходов, в 6 из них получается семь и в 2 получается одиннадцать. Поэтому вероятность успеха p равна $8/36 = 2/9$, и нам в среднем придётся $1/p = 9/2 = 4{,}5$ раза бросить кости, чтобы выпало семь или одиннадцать.

Биномиальное распределение

Рассмотрим n испытаний по схеме Бернулли с вероятностью успеха p и вероятностью неудачи $q = 1 - p$. Пусть случайная величина X — количество успехов в n испытаниях. Её значение может быть равно $0, 1, \ldots, n$, и

$$\mathsf{P}\{X = k\} = C_n^k p^k (1 - p)^{n-k}, \tag{6.36}$$

для любого $k = 0, 1, \ldots, n$ так как имеется C_n^k способов выбрать из n испытаний k удачных, и вероятность каждого такого случая будет $p^k q^{n-k}$. Распределение (6.36) называют **биномиальным** (binomial distribution). Для биномиальных распределений мы используем обозначение

$$b(k; n, p) = C_n^k p^k (1 - p)^{n-k}. \tag{6.37}$$

Пример биномиального распределения показан на рисунке 6.2. Название "биномиальное" связано с тем, что правая часть фор-

Рисунок 6.2 Биномиальное распределение $b(k; 15; 1/3)$, порождаемое $n = 15$ испытаниями по схеме Бернулли, каждое из которых имеет вероятность успеха $p = 1/3$. Математическое ожидание равно $np = 5$.

мулы (6.37) — это k-й член бинома Ньютона $(p+q)^n$. Вспоминая, что $p + q = 1$, получаем

$$\sum_{k=0}^{n} b(k; n, p) = 1, \tag{6.38}$$

как и должно быть (аксиома 2).

Математическое ожидание для случайной величины, имеющей биномиальное распределение, можно вычислить с помощью (6.14) и (6.38). Пусть X — случайная величина, имеющая биномиальное распределение $b(k; n, p)$. Положим $q = 1 - p$. По определению математического ожидания, имеем

$$\begin{aligned}
\mathsf{M}[X] &= \sum_{k=0}^{n} k b(k; n, p) \\
&= \sum_{k=1}^{n} k C_n^k p^k q^{n-k} \\
&= np \sum_{k=1}^{n} C_{n-1}^{k-1} p^{k-1} q^{n-k} \\
&= np \sum_{k=0}^{n-1} C_{n-1}^{k} p^{k} q^{(n-1)-k} \\
&= np \sum_{k=0}^{n-1} b(k; n-1, p) \\
&= np.
\end{aligned} \tag{6.39}$$

Тот же самый результат почти без вычислений можно получить так: пусть X_i — количество успехов в i-м испытании (которое равно 0 с вероятностью q и равно 1 с вероятностью p). Тогда $\mathsf{M}[X_i] = p \cdot 1 + q \cdot 0 = p$. Остаётся заметить, что $X = X_1 + \ldots + X_n$, и потому по свойству линейности (6.26)

$$\mathsf{M}[X] = \mathsf{M}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathsf{M}[X_i] = \sum_{i=1}^{n} p = np.$$

Подобным образом можно вычислить и дисперсию. Из (6.29) следует, что $\mathsf{D}[X_i] = \mathsf{M}[X_i^2] - \mathsf{M}^2[X_i]$. Поскольку X_i принимает лишь значения 0 и 1, то $\mathsf{M}[X_i^2] = \mathsf{M}[X_i] = p$, и, значит,

$$\mathsf{D}[X_i] = p - p^2 = pq. \tag{6.40}$$

Теперь воспользуемся независимостью испытаний и формулой (6.31):

$$\mathsf{D}[X] = \mathsf{D}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathsf{D}[X_i] = \sum_{i=1}^{n} pq = npq. \tag{6.41}$$

На рисунке 6.2 видно, что $b(k; n, p)$ как функция от k сначала увеличивается, пока k не достигнет значения np, а затем уменьшается. Это можно проверить, вычислив отношение двух последовательных членов:

$$\begin{aligned}
\frac{b(k; n, p)}{b(k-1; n, p)} &= \frac{C_n^k p^k q^{n-k}}{C_n^{k-1} p^{k-1} q^{n-k+1}} \\
&= \frac{n!(k-1)!(n-k+1)!p}{k!(n-k)!n!q} \\
&= \frac{(n-k+1)p}{kq} \\
&= 1 + \frac{(n+1)p - k}{kq}.
\end{aligned} \tag{6.42}$$

Это отношение больше 1, когда $(n+1)p - k$ положительно, так что $b(k; n, p) > b(k-1; n, p)$ при $k < (n+1)p$ (функция растёт), и $b(k; n, p) < b(k-1; n, p)$ при $k > (n+1)p$ (функция убывает). Если число $(n+1)p$ — целое, то функция имеет двойной максимум: в точках $(n+1)p$ и $(n+1)p - 1 = np - q$. В противном случае максимум один, и достигается он в целой точке k, лежащей в диапазоне $np - q < k < (n+1)p$.

Следующая лемма даёт верхнюю оценку для биномиального распределения.

Лемма 6.1. *Пусть* $n \geqslant 0$, $0 < p < 1$, $q = 1 - p$, $0 \leqslant k \leqslant n$. *Тогда*

$$b(k; n, p) \leqslant \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}.$$

Доказательство. Согласно неравенству (6.10), имеем

$$\begin{aligned} b(k; n, p) &= C_n^k p^k q^{n-k} \\ &\leqslant \left(\frac{n}{k}\right)^k \left(\frac{n}{n-k}\right)^{n-k} p^k q^{n-k} \\ &= \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}. \end{aligned}$$

□

Упражнения

6.4-1 Проверьте аксиому 2 для геометрического распределения.

6.4-2 Сколько раз в среднем нужно бросать 6 симметричных монет до выпадения 3 орлов и 3 решек (в одном испытании)?

6.4-3 Покажите, что $b(k; n, p) = b(n-k; n, q)$, где $q = 1 - p$.

6.4-4 Покажите, что максимум биномиального распределения $b(k; n, p)$ примерно равен $1/\sqrt{2\pi npq}$, где $q = 1 - p$.

6.4-5* Покажите, что вероятность не получить ни одного успеха в n независимых испытаниях с вероятностью успеха $1/n$ примерно равна $1/e$. Затем покажите, что вероятность получить ровно один успешный исход также приблизительно равна $1/e$.

6.4-6* Профессор бросает симметричную монету n раз, студент делает то же самое. Покажите, что вероятность того, что число орлов у них будет одинаково, равна $C_{2n}^n/4^n$. (Указание. Если считать орла успехом профессора и неудачей студента, то искомое событие есть появление n успехов среди $2n$ испытаний.) Выведите отсюда тождество

$$\sum_{k=0}^{n} (C_n^k)^2 = C_{2n}^n.$$

6.4-7* Покажите, что для $0 \leqslant k \leqslant n$,

$$b(k; n, 1/2) \leqslant 2^{nH(k/n)-n},$$

где $H(x)$ — функция энтропии (6.13).

6.4-8* Рассмотрим n независимых испытаний. Пусть p_i — вероятность успеха в i-м испытании, а случайная величина X есть количество успехов во всех этих испытаниях. Пусть $p \geqslant p_i$ для всех $i = 1, 2, \ldots, n$. Докажите, что

$$\mathsf{P}\{X < k\} \leqslant \sum_{i=0}^{k-1} b(i; n, p)$$

при любом $k = 1, 2, \ldots, n$.

6.4-9* Рассмотрим случайную величину X, равную числу успехов в n испытаниях с вероятностями успеха $p_1, \ldots, p_n$. Пусть X' — аналогичная случайная величина, для которой вероятности успеха равны $p'_1, \ldots, p'_n$. Пусть $p'_i \geqslant p_i$ при всех i. Докажите, что

$$\mathsf{P}\{X' \geqslant k\} \geqslant \mathsf{P}\{X \geqslant k\}.$$

при любом $k = 0, \ldots, n$

(Указание: можно считать, что результаты второй серии испытаний получаются так: сначала делается первая серия, а потом её результаты корректируются случайным образом в сторону увеличения. Используйте результат упражнения 3.3-6.)

6.5 Хвосты биномиального распределения

В оригинале со звездой!

Во многих задачах важна вероятность не в точности k успехов при биномиальном распределении, а *не менее* k успехов (или не более k успехов). В этом разделе мы исследуем этот вопрос, оценив **хвосты** (tails) биномиального распределения. Такие оценки показывают, что большие отклонения числа успехов от математического ожидания (np) маловероятны.

Сначала получим оценку для правого хвоста распределения $b(k; n, p)$. Оценки для левого хвоста симметричны (успехи меняются местами с неудачами).

Теорема 6.2. *Пусть X — число успехов в серии из n независимых испытаний с вероятностью успеха p. Тогда*

$$\mathsf{P}\{X \geqslant k\} = \sum_{i=k}^{n} b(i; n, p) \leqslant C_n^k p^k.$$

Доказательство. Воспользуемся неравенством (6.15):

$$C_n^{k+i} \leqslant C_n^k C_{n-k}^i.$$

Заметим, что

$$\begin{aligned}
\mathsf{P}\{X \geqslant k\} &= \sum_{i=k}^{n} b(i; n, p) \\
&= \sum_{i=0}^{n-k} b(k+i; n, p) \\
&= \sum_{i=0}^{n-k} C_n^{k+i} p^{k+i} (1-p)^{n-(k+i)} \\
&\leqslant \sum_{i=0}^{n-k} C_n^k C_{n-k}^i p^{k+i} (1-p)^{n-(k+i)} \\
&= C_n^k p^k \sum_{i=0}^{n-k} C_{n-k}^i p^i (1-p)^{(n-k)-i} \\
&= C_n^k p^k \sum_{i=0}^{n-k} b(i; n-k, p) \\
&= C_n^k p^k,
\end{aligned}$$

так как $\sum_{i=0}^{n-k} b(i; n-k, p) = 1$ по формуле (6.38). □

Переписывая это утверждение для левого хвоста, получаем такое

Следствие 6.3. Пусть X — число успехов в серии из n независимых испытаний с вероятностью успеха p. Тогда

$$\mathsf{P}\{X \leqslant k\} = \sum_{i=0}^{k} b(i; n, p) \leqslant C_n^{n-k} (1-p)^{n-k} = C_n^k (1-p)^{n-k}.$$

Наша следующая оценка будет для левого хвоста биномиального распределения: с удалением границы от точки максимума вероятность, приходящаяся на хвост, экспоненциально уменьшается.

Теорема 6.4. *Пусть X — число успехов в серии из n независимых испытаний с вероятностью успеха p и вероятностью неудачи $q = 1 - p$. Тогда*

$$\mathsf{P}\{X < k\} = \sum_{i=0}^{k-1} b(i; n, p) < \frac{kq}{np-k} b(k; n, p)$$

при $0 < k < np$.

Доказательство. Мы сравниваем $\sum_{i=0}^{k-1} b(i; n, p)$ с суммой геометрической прогрессии (см. разд. 3.2). Для $i = 1, 2, \ldots, k$ формула (6.42) даёт

$$\frac{b(i-1; n, p)}{b(i; n, p)} = \frac{iq}{(n-i+1)p} < \left(\frac{i}{n-i}\right)\left(\frac{q}{p}\right) \leqslant \left(\frac{k}{n-k}\right)\left(\frac{q}{p}\right).$$

Положив
$$x = \left(\frac{k}{n-k}\right)\left(\frac{q}{p}\right) < 1,$$
видим, что в последовательности $b(i; n, p)$ при $0 \leqslant i \leqslant k$ каждый член меньше следующего, умноженного на x. Поэтому интересующая нас сумма (от $i = 0$ до $i = k - 1$) меньше значения $b(k; n, p)$, умноженного на $x + x^2 + x^3 + \ldots = x/(1 - x)$:
$$\sum_{i=0}^{k-1} b(i; n, p) < \frac{x}{1-x} b(k; n, p) = \frac{kq}{np-k} b(k; n, p).$$

□

При $k \leqslant np/2$ коэффициент $kq/(np - k)$ не превосходит 1, так что $b(k; n, p)$ является оценкой сверху для суммы всех предыдущих членов. Для примера рассмотрим n бросаний симметричной монеты. Положим $p = 1/2$ и $k = n/4$. Теорема 6.4 гарантирует, что вероятность появления менее чем $n/4$ орлов меньше вероятности выпадения в точности $n/4$ орлов. (Более того, для любого положительного $r \leqslant n/4$ вероятность появления менее r орлов меньше вероятности появления в точности r орлов.) Теорема 6.4 может быть использована вместе с оценками биномиального распределения сверху, например с леммой 6.1.

Симметричная оценка для правого хвоста выглядит так:

Следствие 6.5. Пусть X — число успехов в серии из n независимых испытаний с вероятностью успеха p и вероятностью неудачи $q = 1 - p$. Тогда
$$\mathsf{P}\{X > k\} = \sum_{i=k+1}^{n} b(i; n, p) < \frac{(n-k)p}{k-np} b(k; n, p)$$
при $np < k < n$.

В следующей теореме рассматривается более общий случай: каждое из испытаний имеет свою вероятность успеха.

Теорема 6.6. *Рассмотрим серию из n независимых испытаний; вероятность успеха в i-м из них обозначим p_i (вероятность неудачи q_i равна $1 - p_i$). Пусть случайная величина X есть число успехов в серии, и пусть $\mu = \mathsf{M}[X]$. Тогда для $r > \mu$ выполнено неравенство*
$$\mathsf{P}\{X - \mu \geqslant r\} \leqslant \left(\frac{\mu e}{r}\right)^r.$$

Доказательство. Для любого $\alpha > 0$ функция $e^{\alpha x}$ строго возрастает по x, поэтому
$$\mathsf{P}\{X - \mu \geqslant r\} = \mathsf{P}\{e^{\alpha(X-\mu)} \geqslant e^{\alpha r}\},$$

Для оценки правой части мы используем неравенство Маркова (6.32) (а наиболее выгодное значение α подберём позднее):

$$\mathsf{P}\{X-\mu \geqslant r\} \leqslant \mathsf{M}[e^{\alpha(X-\mu)}]e^{-\alpha r}. \tag{6.43}$$

Остаётся оценить $\mathsf{M}[e^{\alpha(X-\mu)}]$ и выбрать значение для α. Рассмотрим случайную величину X_i, равную 1 в случае успеха i-го испытания и 0 в случае его неудачи. Тогда

$$X=\sum_{i=1}^{n} X_i$$

и

$$X-\mu=\sum_{i=1}^{n}(X_i-p_i).$$

Поскольку испытания независимы, то величины X_i независимы. Поэтому величины $e^{\alpha(X_i-p_i)}]$ независимы (упр. 6.3-4), и по формуле (6.27) можно переставить произведение и математическое ожидание:

$$\mathsf{M}[e^{\alpha(X-\mu)}]=\mathsf{M}\left[\prod_{i=1}^{n} e^{\alpha(X_i-p_i)}\right]=\prod_{i=1}^{n}\mathsf{M}[e^{\alpha(X_i-p_i)}].$$

Каждый множитель можно оценить так:

$$\begin{aligned}
\mathsf{M}[e^{\alpha(X_i-p_i)}] &= e^{\alpha(1-p_i)}p_i+e^{\alpha(0-p_i)}q_i\\
&= p_i e^{\alpha q_i}+q_i e^{-\alpha p_i}\\
&\leqslant p_i e^{\alpha}+1 \qquad (6.44)\\
&\leqslant \exp(p_i e^{\alpha}),
\end{aligned}$$

где $\exp(x)$ обозначает экспоненциальную функцию: $\exp(x)=e^x$. (Мы воспользовались тем, что $\alpha>0$, $q_i\leqslant 1$, $e^{\alpha q_i}\leqslant e^{\alpha}$ и $e^{-\alpha p_i}\leqslant 1$, а также неравенством (2.7).) Следовательно,

$$\mathsf{M}[e^{\alpha(X-\mu)}]\leqslant\prod_{i=1}^{n}\exp(p_i e^{\alpha})=\exp(\mu e^{\alpha}),$$

так как $\mu=\sum_{i=1}^{n} p_i$. Таким образом, из неравенства (6.43) следует оценка

$$\mathsf{P}\{X-\mu\geqslant r\}\leqslant\exp(\mu e^{\alpha}-\alpha r). \tag{6.45}$$

Выбирая $\alpha=\ln(r/\mu)$ (см. упр. 6.5-6), получаем

$$\begin{aligned}
\mathsf{P}\{X-\mu\geqslant r\} &\leqslant \exp(\mu e^{\ln(r/\mu)}-r\ln(r/\mu))\\
&= \exp(r-r\ln(r/\mu))=\frac{e^r}{(r/\mu)^r}=\left(\frac{\mu e}{r}\right)^r.
\end{aligned}$$

$\square$

[Формально эта оценка применима при любом $r > \mu$, но её имеет смысл применять только если r больше μ более чем в e раз, иначе правая часть будет больше единицы.]

Теорему 6.6 можно применять и для случая равных вероятностей. При этом $\mu = \mathsf{M}[X] = np$, и получается такое

Следствие 6.7. Пусть X — число успехов в серии из n независимых испытаний по схеме Бернулли с вероятностью успеха p. Тогда

$$\mathsf{P}\{X - np \geqslant r\} = \sum_{k=\lceil np+r \rceil}^{n} b(k;n,p) \leqslant \left(\frac{npe}{r}\right)^r .$$

Упражнения

6.5-1* Что менее вероятно: не получить ни одного орла при n бросаниях симметричной монеты, или получить менее n орлов при $4n$ бросаниях симметричной монеты?

6.5-2* Покажите, что

$$\sum_{i=0}^{k-1} C_n^i a^i < (a+1)^n \frac{k}{na - k(a+1)} b(k;n,a/(a+1))$$

при $a > 0$ и $0 < k < n$.

6.5-3* Докажите, что при $0 < k < np$, где $0 < p < 1$ и $q = 1 - p$, выполнено неравенство

$$\sum_{i=0}^{k-1} p^i q^{n-i} < \frac{kq}{np-k} \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k} .$$

6.5-4* Покажите, что в условиях теоремы 6.6 выполнено неравенство

$$\mathsf{P}\{\mu - X \geqslant r\} \leqslant \left(\frac{(n-\mu)e}{r}\right)^r ,$$

а в условиях следствия 6.7 выполнено неравенство

$$\mathsf{P}\{np - X \geqslant r\} \leqslant \left(\frac{nqe}{r}\right)^r .$$

6.5-5* Рассмотрим серию из n независимых испытаний; вероятность успеха в i-м из них обозначим p_i (вероятность неудачи q_i равна $1 - p_i$). Пусть случайная величина X есть число успехов в серии, и пусть $\mu = \mathsf{M}[X]$. Докажите, что при $r \geqslant 0$ выполнено неравенство

$$\mathsf{P}\{X - \mu \geqslant r\} \leqslant e^{-r^2/2n}.$$

6.5-6* Покажите, что при выбранном значении $\alpha = \ln(r/\mu)$ правая часть неравенства (6.45) достигает минимума.

6.6 Вероятностный анализ

В этом разделе мы приведём три примера применения разобранных методов оценки вероятностей. Примеры эти таковы: совпадение дней рождений у двух человек среди данных k человек, распределение шаров по урнам и участки повторяющихся исходов при бросании монеты.

6.6.1 Парадокс дня рождения

Парадокс дня рождения (birthday paradox) связан с таким вопросом: сколько человек должно быть в комнате, чтобы с большой вероятностью среди них оказались двое родившихся в один день? Парадокс состоит в том, что ответ значительно меньше числа дней в году, что кажется странным.

Мы считаем, что в году 365 дней и что дни рождения k человек выбираются случайно и независимо друг от друга. Оценим вероятность того, что все дни рождения окажутся различными. Пусть день рождения первого уже выбран; ясно, что день рождения второго совпадёт с ним с вероятностью 1/365. При выбранных (и различных) днях рождения первого и второго вероятность, что у третьего день рождения совпадёт с одним из уже имеющихся, будет 2/365 и так далее. В итоге вероятность того, что у k человек будут различные дни рождения, есть

$$\left(1-\frac{1}{365}\right)\left(1-\frac{2}{365}\right)\dots\left(1-\frac{k-1}{365}\right).$$

Более формально, пусть n — число дней в году, и пусть A_i — событие *"день рождения $(i+1)$-го человека не совпадает с днями рождения предыдущих i человек"*. Тогда пересечение $B_i = A_1 \cap A_2 \cap \ldots \cap A_{i-1}$ будет событием *"у первых i человек дни рождения различны"*.

Поскольку $B_k = A_{k-1} \cap B_{k-1}$, то из формулы (6.20) получаем соотношение

$$\mathrm{P}\{B_k\} = \mathrm{P}\{B_{k-1}\}\mathrm{P}\{A_{k-1}|B_{k-1}\}. \qquad (6.46)$$

Начальное условие: $\mathrm{P}\{B_1\} = 1$.

Условная вероятность $\mathrm{P}\{A_{k-1}|B_{k-1}\}$ равна $(n-k+1)/n$, так как среди n дней имеется $n-(k-1)$ свободных (по условию все предыдущие дни рождения различны). Поэтому

$$\begin{aligned}\mathsf{P}\{B_k\} &= \mathsf{P}\{B_1\}\mathsf{P}\{A_1|B_1\}\mathsf{P}\{A_2|B_2\}\cdots\mathsf{P}\{A_{k-1}|B_{k-1}\}\\ &= 1\cdot\left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right)\cdots\left(\frac{n-k+1}{n}\right)\\ &= 1\cdot\left(1-\frac{1}{n}\right)\left(1-\frac{2}{n}\right)t\ldots\left(1-\frac{k-1}{n}\right)\end{aligned}$$

Теперь из неравенства $1+x \leqslant e^x$ (2.7) следует, что

$$\begin{aligned}\mathsf{P}\{B_k\} &\leqslant e^{-1/n}e^{-2/n}e^{-(k-1)/n}\\ &= e^{-(1+2+3+\ldots+(k-1))/n}\\ &= e^{-k(k-1)/2n}\\ &\leqslant 1/2,\end{aligned}$$

если $-k(k-1)/2n \leqslant ln(1/2)$. Вероятность того, что все k дней рождений различны, не превосходит $1/2$ при $k(k-1) \geqslant 2n\ln 2$. Решая это квадратное неравенство, получаем $k \geqslant (1+\sqrt{1+(8\ln 2)n})/2$. Для $n = 365$, имеем $k \geqslant 23$. Итак, если в комнате находится не менее 23 человек, то с вероятностью не менее 1/2 какие-то двое из них родились в один и тот же день. На Марсе, где год состоит из 669 марсианских суток, в комнате должно быть не менее 31 марсианина.

Другой метод анализа

Есть другой, более простой способ получить оценку для родственной задачи. Для каждой пары людей (i, j), находящихся в комнате, рассмотрим случайную величину X_{ij}

$$X_{ij} = \begin{cases} 1, & \text{если } i \text{ и } j \text{ родились в один день,}\\ 0, & \text{в противном случае.}\end{cases}$$

Вероятность того, что дни рождения двух данных людей совпадают, равна $1/n$, поэтому по определению математического ожидания (6.23)

$$\mathsf{M}[X_{ij}] = 1\cdot(1/n) + 0\cdot(1-1/n) = 1/n$$

при $i \neq k$.

Сумма всех X_{ij} по всем парам $1 \leqslant i < j \leqslant k$ имеет математическое ожидание, равное сумме ожиданий для каждой пары; всего пар $C_k^2 = k(k-1)/2$, так что эта сумма равна $k(k-1)/2n$. Поэтому нужно примерно $\sqrt{2n}$ человек, чтобы математическое ожидание числа пар людей с совпадающими днями рождения сравнялось с 1.

Например, при $n = 365$ и $k = 28$ ожидаемое число пар людей, родившихся в один день, равно $(28 \cdot 27)/(2 \cdot 365) \approx 1{,}0356$. На Марсе для этого требуется 38 марсиан.

Заметим, что мы оценивали две разные вещи: (1) при каком k вероятность события $X = \sum X_{ij} > 0$ больше $1/2$, и (2) при каком k математическое ожидание X больше 1. Формально это разные вопросы (можно заметить лишь, что если вероятность $\mathsf{P}\{X > 0\} > 1/2$, то $\mathsf{M}[X] > 1/2$, так как величина X принимает целые значения). Однако и в том, и в другом случае ответ имеет асимптотику $\Theta(\sqrt{n})$.

6.6.2 Шары и урны

Пусть имеется b урн, пронумерованных от 1 до b. Мы опускаем в них шары: каждый шар с равной вероятностью помещается в одну из урн независимо от предыдущих. Таким образом, с точки зрения любой из урн происходит последовательность испытаний по схеме Бернулли с вероятностью успеха $1/b$ (успех — попадание шара в эту урну). Мы рассмотрим несколько задач, связанных с таким процессом.

Сколько шаров попадёт в данную урну? Количество шаров, попавших в данную урну, описывается биномиальным распределением $b(k; n, 1/b)$. Если всего бросается n шаров, то математическое ожидание числа попавших в урну шаров равно n/b.

Сколько в среднем шаров нужно бросить, пока в данную урну не попадёт шар? Количество бросков до первого попадания в заданную урну имеет геометрическое распределение с вероятностью $1/b$, поэтому ожидаемое число бросков есть $1/(1/b) = b$.

Сколько шаров нужно бросить, чтобы каждая урна содержала по меньшей мере один шар? Будем следить за числом заполненных урн. Вначале оно равно нулю, а затем увеличивается, пока не достигнет b. Обозначим через n_i случайную величину, равную числу попыток, потребовавшихся, чтобы это число возросло от $i-1$ до i. (Таким образом, если вторая и третья попытки пришлись на ту же урну, что первая, а четвёртая — на другую, то $n_1 = 1$, $n_2 = 3$.) Общее число попыток до заполнения всех урн равно $n_1 + n_2 + \ldots + n_b$. Мы вычислим математическое ожидание этой суммы как сумму математических ожиданий. Когда мы ждём заполнения i-й урны, заполнено $i-1$ урн из b и вероятность попасть в незаполненную равна $(b-i+1)/b$. По свойству геометрического распределения математическое ожидание величины n_i обратно этой вероятности и равно $b/(b-i+1)$. Сумма этих величин по всем i равна $b(1/b + 1/(b-1) + \ldots + 1/2 + 1) = b(\ln b + O(1))$. (См. формулу (3.5) для суммы гармонического ряда.)

Итак, требуется сделать в среднем примерно $b \ln b$ бросков, прежде чем в каждой урне появится по шару.

6.6.3 Участки повторяющихся исходов

Пусть мы бросаем симметричную монету n раз. Какое максимальное число идущих подряд орлов мы ожидаем увидеть? Оказывается, ответ на этот вопрос — $\Theta(\lg n)$.

Сначала докажем, что ожидаемая длина наибольшего участка есть $O(\lg n)$. Пусть событие A_{ik} состоит в том, что имеется участок из k или более орлов, начинающийся с i-го бросания. Очевидно,

$$\mathsf{P}\{A_{ik}\} = 1/2^k. \tag{6.47}$$

При $k = 2\lceil \lg n \rceil$ вероятность появления k орлов в данных позициях не превосходит $1/n^2$, а возможных мест (значений i) меньше чем n, так что вероятность появления k орлов подряд (где-нибудь) не больше $1/n$. Теперь математическое ожидание максимального числа идущих подряд орлов оценивается так: это число никогда не превосходит n и почти всегда (с вероятностью $1-1/n$) не превосходит $2\lceil \lg n \rceil$, поэтому ожидание не больше $\lceil 2\lg n \rceil + n\cdot(1/n) = O(\lg n)$.

Вероятность образования участка орлов длины не менее $r\lceil \lg n \rceil$ быстро уменьшается с ростом r (для фиксированной позиции она не больше $2^{-r\lg n} = n^{-r}$, а для всех позиций в сумме она не превосходит $n \cdot n^{-r} = n^{-(r-1)}$.) Например, для $n = 1000$ вероятность появления участка из по меньшей мере $2\lceil \lg n \rceil = 20$ орлов не превосходит $1/n = 1/1000$, а вероятность появления участка $3\lceil \lg n \rceil = 30$ орлов не больше $1/n^2 = 10^{-6}$.

Теперь докажем оценку снизу: ожидаемая длина наибольшего участка есть $\Omega(\lg n)$. Для этого рассмотрим участки длины $\lfloor (\lg n)/2 \rfloor$. Согласно (6.47) вероятность появления такого участка в данной позиции не меньше $1/2^{\lfloor (\lg n)/2 \rfloor} \geqslant 1/\sqrt{n}$, а вероятность его непоявления не больше $1 - 1/\sqrt{n}$. Разобьём всю последовательность бросаний на непересекающиеся группы, состоящие из $\lfloor (\lg n)/2 \rfloor$ бросаний каждая. (Несколько членов окажутся вне групп, если при делении будет остаток.) Число групп не меньше $2n/\lg n - 1$.

События в разных группах независимы, поэтому вероятность того, что ни одна из этих групп не состоит из одних орлов, не больше

$$\begin{aligned}(1 - 1/\sqrt{n})^{2n/\lg n - 1} &\leqslant e^{-(2n/\lg n - 1)/\sqrt{n}} \\ &= O(e^{-\lg n}) = O(1/n)\end{aligned}$$

Мы использовали тот факт, что $1 + x \leqslant e^x$ (2.7), а также то, что $(2n/\lg n - 1)/\sqrt{n} \geqslant \lg n - O(1)$.

Итак, с вероятностью не менее $1 - O(1/n)$ длина наибольшего участка подряд идущих орлов не меньше $\lfloor \lg n/2 \rfloor = \Omega(\lg n)$, поэтому математическое ожидание никак не меньше $(1-1/n)\Omega(\lg n) = \Omega(\lg n)$.

Упражнения

6.6-1 Шары бросают в b урн; все бросания независимы друг от друга, и каждый шар равновероятно попадает в любую из урн. Чему равно ожидаемое количество бросаний до момента, когда в одной из урн окажется два шара?

6.6-2* Существенно ли для приведённого нами анализа парадокса дня рождения то, что дни рождения независимы в совокупности, или было бы достаточно их попарной независимости? Объясните свой ответ.

6.6-3* Скольких гостей надо пригласить на вечеринку, чтобы скорее всего оказалось, что по меньшей мере *трое* из них родились в один день?

6.6-4* Какую долю составляют инъекции среди всех отображений k-элементного множества в n-элементное? Как связан этот вопрос с парадоксом дня рождения?

6.6-5* Пусть n шаров бросают в n урн, все бросания независимы, попадания каждого шара во все урны равновероятны. Чему равно ожидаемое количество пустых урн? А ожидаемое количество урн, в которые попало в точности по одному шару?

6.6-6* Улучшите нижнюю оценку для длин участков из одних орлов, показав, что при n бросаниях симметричной монеты участок длины $\lg n - 2\lg\lg n$ найдётся с вероятностью не меньше $1 - 1/n$.

Задачи

6-1 Шары и урны

В этой задаче мы считаем число способов разложить n шаров в b различных урн.

а. Предположим, что все шары разные, их порядок внутри урны не учитывается. Докажите, что существует b^n способов поместить шары в урны.

б. Предположим, что все шары разные и их порядок в урне существен. Докажите, что шары можно разложить по урнам $(b+n-1)!/(b-1)!$ способами. (Указание: подсчитайте число способов расставить n различных шаров и $b-1$ одинаковых чёрточек в ряд.)

в. Предположим, что все шары одинаковы, и их порядок в урне не имеет значения. Покажите, что число способов раскладки шаров по урнам равняется C^n_{b+n-1}. (Указание: используйте ту же идею, что в пункте (б).)

г. Покажите, что если шары одинаковые и в любую урну помещается не больше одного шара, то число способов равно C^n_b.

д. Покажите, что если шары одинаковые и в любой урне должен оказаться по меньшей мере один шар, то число способов раскладки шаров равно C^{b-1}_{n-1}.

6-2 Программа вычисления максимума

Рассмотрим такую программу поиска максимума в неупорядоченном массиве $A[1\,..\,n]$.

```
1 $max \leftarrow -\infty$
2 for $i \leftarrow 1$ to $n$
3     do ▷ Сравнить $A[i]$ с $max$.
4        if $A[i] > max$
5           then $max \leftarrow A[i]$
```

Мы хотим определить, сколько раз в среднем выполняется присваивание в строке 5. Предполагается, что числа в массиве A различны и расположены в случайном порядке (все перестановки равновероятны).

а. Если число x случайно выбрано из i различных чисел, то с какой вероятностью оно окажется максимальным числом среди них?

б. Как соотносится $A[i]$ с предыдущими элементами массива для тех i, при которых выполняется строка 5?

в. Чему равна вероятность выполнения строки 5 программы для данного значения i?

г. Пусть s_i — случайная величина, равная 1 или 0 в зависимости от того, выполнялась строка 5 на i-м шаге цикла или нет. Чему равно $\mathsf{M}[s_i]$?

д. Пусть $s = s_1 + s_2 + \cdots + s_n$ — общее число присваиваний в строке 5 при исполнении всей программы. Покажите, что $\mathsf{M}[s] = \Theta(\lg n)$.

6-3 Проблема выбора

Заведующая кафедрой принимает на работу нового сотрудника. Она назначила собеседования n претендентам и хочет выбрать наиболее квалифицированного из них. Однако университетские правила требуют, чтобы после беседы претенденту сразу сообщалось, принят он или нет.

Для этого она применяет такое правило. Сначала она говорит с первыми k претендентами, отказывая им независимо от их квалификации. Если среди оставшихся есть более квалифицированный, чем первые k, то первый из таковых принимается. Если нет, принимается последний из претендентов. Покажите, что вероятность выбрать таким способом лучшего из претендентов будет максимальна (и равна примерно $1/e$), если k примерно равно n/e.

6-4 Вероятностный счётчик

С помощью t-битного счётчика мы можем считать до $2^t - 1$. Следующий приём **вероятностного подсчёта** (probabilistic counting, R. Morris) даёт возможность вести счёт до куда больших значений, правда ценою некоторой потери точности.

Выберем возрастающую последовательность целых неотрицательных чисел n_i (где i меняется от 0 до 2^t-1). Её смысл таков: если значение t-битного регистра равно i, то это означает, что подсчитываемое количество (число выполненных операций INCREMENT) примерно равно n_i. Мы считаем, что $n_0 = 0$.

Операция INCREMENT увеличивает значение счетчика, содержащего i, с некоторой вероятностью. Именно, число i увеличивается на 1 с вероятностью $1/(n_{i+1} - n_i)$, и остается неизменным в остальных случаях. (Если $i = 2^t - 1$, то происходит переполнение.) Идея проста: в среднем для увеличения i на единицу потребуется как раз $n_{i+1} - n_i$ операций.

Если $n_i = i$ для всех $i \geqslant 0$, то получаем обычный счётчик. Более интересные ситуации возникают, если выбрать, например, $n_i = 2^{i-1}$ для $i > 0$ или $n_i = F_i$ (i-е число Фибоначчи, см. разд. 2.2).

Мы будем предполагать, что n_{2^t-1} достаточно велико, и пренебрегать возможностью переполнения.

а. Покажите, что математическое ожидание содержимого счётчика после выполнения n операций INCREMENT, в точности равно n.

б. Дисперсия случайной величины, равной содержимому счётчика после n операций INCREMENT, зависит от выбора последовательности $n_0, n_1, \ldots$. Найдите эту дисперсию для случая $n_i = 100i$.

Замечания

Общие методы решения вероятностных задач обсуждались в знаменитой переписке Паскаля (B. Pascal) и Ферма (P. de Fermat), начавшейся в 1654 году, и в книге Гюйгенса (C. Huygens, 1657). Более строгое изложение теории вероятностей было дано в работах Бернулли (J. Bernoulli, 1713) и Муавра (A. De Moivre, 1730). Дальнейшее развитие теории вероятностей связано с именами Лапласа (P. S. de Laplace), Пуассона (S.-D. Poisson) и Гаусса (C. F. Gauss).

Суммы случайных величин исследовались П. Л. Чебышёвым и А. А. Марковым (старшим). В 1933 году А. Н. Колмогоров сформулировал аксиомы теории вероятностей. Оценки хвостов распределений приводят Чернов [40] и Хофдинг [99]. Важные результаты о случайных комбинаторных структурах принадлежат Эрдёшу (P. Erdös).

Литература по теме главы: Кнут [121], Лю [140] (хорошие пособия по элементарной комбинаторике и подсчету); Биллингсли [28], Чанг [41], Дрейк [57], Феллер [66], Розанов [171] (стандартные учебники по теории вероятностей); Боллобас [30], Хофри [100], Спенсер [179] (техника вероятностного анализа).

II Сортировка и порядковые статистики

Введение

В этой части мы рассмотрим несколько алгоритмов, решающих **задачу сортировки** (sorting problem). Исходным данным для этой задачи является последовательность чисел $\langle a_1, a_2, \ldots, a_n \rangle$. Результатом должна быть последовательность $\langle a'_1, a'_2, \ldots, a'_n \rangle$, состоящая из тех же чисел, идущих в неубывающем порядке: $a'_1 \leqslant a'_2 \leqslant \cdots \leqslant a'_n$. Обычно исходная последовательность задана как массив, хотя возможны и другие варианты (например, связанный список).

Структура сортируемых объектов

На практике редко требуется упорядочивать числа как таковые. Обычно надо сортировать **записи** (records), содержащие несколько полей, и располагать их в порядке, определяемым одним из полей. [Например, в архиве отдела кадров для каждого сотрудника фирмы может храниться запись, содержащая различные поля (фамилия, имя, отчество, год рождения, адрес и т. п.), и в какой-то момент может понадобиться упорядочить все записи по годам рождения.] Поле, по которому проводится сортировка (год рождения в нашем примере), называется **ключом** (key), а остальные поля — дополнительными данными (satellite data). Можно представлять себе дело так: алгоритм сортирует ключи, но вместе с каждым ключом перемещаются (без изменения) дополнительные данные, с ним связанные. (Если этих данных много, разумно перемещать не сами данные, а лишь указатель на них.)

Все эти подробности мы не рассматриваем, ограничиваясь задачей сортировки ключей. Приводимые нами алгоритмы являются, таким образом, лишь *"скелетом"* реальной программы, к которому нужно добавить обработку дополнительных данных (что обычно не сложно, хотя и хлопотно).

Алгоритмы сортировки

Мы уже встречались в главе 1 с двумя алгоритма, сортирующими n чисел за время $\Theta(n^2)$ в худшем случае. Их простота, однако, делает их эффективными для сортировки небольшого массива без дополнительной памяти (in place). (Имеется в виду, что мы не имеем права заводить ещё одного массива, но переменные для чисел использовать можно). Алгоритм сортировки слиянием имеет лучшую асимптотическую оценку времени работы, но он требует дополнительного массива.

В этой части книги мы рассмотрим ещё два алгоритма сортировки вещественных чисел. Первый из них называется *"сортировкой с помощью кучи"* (heapsort). Здесь *"куча"* — некоторая структура данных, имеющая много приложений (одно из них, очереди с приоритетами, мы также рассмотрим в главе 7). Этот алгоритм не требует дополнительного массива и работает за время $\Theta(n \lg n)$ в худшем случае.

В главе 8 строится другой алгоритм, называемый *"быстрой сортировкой"* (quicksort). В худшем случае он требует времени $\Theta(n^2)$, но в среднем — лишь $\Theta(n \lg n)$, и на практике он обычно быстрее сортировки с помощью кучи (его внутренний цикл прост, поэтому константа в асимптотической оценке меньше).

Все эти алгоритмы (сортировки вставками, слиянием, с помощью кучи и быстрая сортировка) используют только попарные сравнения объектов, но не их внутреннюю структуру. В главе 9 мы показываем, что любая сортировка такого вида в худшем случае требует времени $\Omega(n \lg n)$, введя подходящую формальную модель (разрешающие деревья). Тем самым становится ясным, что сортировки слиянием и с помощью кучи асимптотически оптимальны.

Однако эта нижняя оценка не распространяется на алгоритмы, использующие внутреннюю структуру данных. В главе 9 рассматривается несколько примеров такого рода. Сортировка подсчётом (counting sort) позволяет отсортировать n целых чисел в диапазоне от 1 до k за время $O(n+k)$, используя сортируемые числа как индексы в массиве размера k. Если $k = O(n)$, время сортировки становится пропорциональным размеру массива. Родственный алгоритм, называемый *"цифровой сортировкой"*, применяет сходный приём поразрядно, и позволяет отсортировать n целых чисел, каждое из которых имеет d разрядов в k-ичной системе счисления, за время $O(d(n+k))$. Если считать, что d постоянно, а k есть $O(n)$, то общее время есть $O(n)$. Третий алгоритм такого рода, сортировка вычёрпыванием, предполагает, что сортируемые числа равномерно распределены на отрезке и независимы, и в среднем требует $O(n)$ действий для n чисел.

Порядковые статистики

Если нам надо найти i-й по величине элемент массива, можно сначала отсортировать массив, что требует времени $\Omega(n \lg n)$, если не пользоваться внутренней структурой элементов (глава 9). Но при этом выполняется лишняя работа (сортировка остальных элементов), и можно построить более эффективные алгоритмы для задачи отыскания i-го по величине элемента (которую называют также задачей о порядковых статистиках). В главе 10 приводятся два таких алгоритма: один требует времени $O(n^2)$ в худшем случае и $O(n)$ в среднем; другой, более сложный, обходится временем $O(n)$ в худшем случае.

Используемые сведения из математики

Большая часть материала глав 7–10 доступна читателю с минимальной математической подготовкой. Однако вероятностный анализ алгоритмов (быстрой сортировки, сортировки вычёрпыванием и отыскания i-го по величине элемента) требует знакомства с теорией вероятностей (глава 6). Добавим, что алгоритм отыскания i-го элемента за линейное время (в худшем случае) более сложен, чем другие алгоритмы этой главы.

7 Сортировка с помощью кучи

В этой главе рассмотрен алгоритм **сортировки с помощью кучи** (heapsort). Как и алгоритм сортировки слиянием, он требует времени $O(n \lg n)$ для сортировки n объектов, но обходится дополнительной памятью размера $O(1)$ (вместо $O(n)$ для сортировки слиянием). Таким образом, этот алгоритм сочетает преимущества двух ранее рассмотренных алгоритмов — сортировки слиянием и сортировки вставками.

Структура данных, которую использует алгоритм (она называется ”*двоичной кучей*”) оказывается полезной и в других ситуациях. В частности, на её базе можно эффективно организовать очередь с приоритетами (см. разд. 7.5). В следующих главах нам встретятся алгоритмы, использующие сходные структуры данных (биномиальные кучи, фибоначчиевы кучи).

Термин ”*куча*” иногда используют в другом смысле (область памяти, где данные размещаются с применением автоматической ” *сборки мусора*” — например, в языке Lisp), но мы этого делать не будем.

7.1 Кучи

Двоичной кучей (binary heap) называют массив с определёнными свойствами упорядоченности, Чтобы сформулировать эти свойства, будем рассматривать массив как двоичное дерево (рис. 7.1). Каждая вершина дерева соответствует элементу массива. Если вершина имеет индекс i, то её родитель имеет индекс $\lfloor i/2 \rfloor$ (вершина с индексом 1 является корнем), а её дети — индексы $2i$ и $2i+1$. Будем считать, что куча может не занимать всего массива и хранить массив A, его длину $length[A]$ и специальный параметр $heap\text{-}size[A]$ (размер кучи), причём $heap\text{-}size[A] \leqslant length[A]$. Куча состоит из элементов $A[1], \ldots, A[heap\text{-}size[A]]$. Движение по дереву осуществляется процедурами

Рисунок 7.1 Кучу можно рассматривать как дерево (а) или как массив (б). Внутри вершины показано её значение. Около вершины показан её индекс в массиве.

```
Parent($i$)
    return $\lfloor i/2 \rfloor$

Left($i$)
    return $2i$

Right($i$)
    return $2i + 1$
```

Элемент $A[1]$ является корнем дерева.

В большинстве компьютеров для выполнения процедур Left и Parent можно использовать команды левого и правого сдвига (Left, Parent); Right требует левого сдвига, после которого в младший разряд помещается единица.

Элементы, хранящиеся в куче, должны обладать **основным свойством кучи** (heap property): для каждой вершины i, кроме корня (т.е. при $2 \leqslant i \leqslant \textit{heap-size}[A]$),

$$A[\text{Parent}(i)] \geqslant A[i]. \tag{7.1}$$

Отсюда следует, что значение потомка не превосходит значения предка. Таким образом, наибольший элемент дерева (или любого поддерева) находится в корневой вершине дерева (этого поддерева).

Высотой (height) вершины дерева называется высота поддерева с корнем в этой вершине (число рёбер в самом длинном пути с началом в этой вершине вниз по дереву к листу). Высота дерева, таким образом, совпадает с высотой его корня. В дереве, составляющем кучу, все уровни (кроме, быть может, последнего), заполнены полностью. Поэтому высота этого дерева равна $\Theta(\lg n)$, где n — число элементов в куче (см. упр. 7.1-2). Как мы увидим ниже, время работы основных операций над кучей пропорционально высоте дерева и, следовательно, составляет $O(\lg n)$. Оставшаяся часть главы посвящена анализу этих операций и применению кучи в задачах сортировки и моделирования очереди с приоритетами. Перечислим основные операции над кучей:

- Процедура Heapify позволяет поддерживать основное свойство (7.1). Время работы составляет $O(\lg n)$.
- Процедура Build-Heap строит кучу из исходного (неотсортированного) массива. Время работы $O(n)$.
- Процедура Heapsort сортирует массив, не используя дополнительной памяти. Время работы $O(n \lg n)$.
- Процедуры Extract-Max (взятие наибольшего) и Insert (добавление элемента) используются при моделировании очереди с приоритетами на базе кучи. Время работы обеих процедур составляет $O(\lg n)$.

Упражнения

7.1-1 Пусть куча имеет высоту h. Сколько элементов может в ней быть? (Укажите максимальное и минимальное значения.)

7.1-2 Докажите, что куча из n элементов имеет высоту $\lfloor \lg n \rfloor$.

7.1-3 Докажите, что при выполнении основного свойства кучи корневая вершина любого поддерева является наибольшей в этом поддереве.

7.1-4 Где может находиться наименьший элемент кучи, если все её элементы различны?

7.1-5 Пусть массив отсортирован в обратном порядке (первый элемент — наибольший). Является ли такой массив кучей?

7.1-6 Является ли кучей массив $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$?

7.2 Сохранение основного свойства кучи

Процедура Heapify — важное средство работы с кучей. Её параметрами являются массив A и индекс i. Предполагается, что поддеревья с корнями Left(i) и Right(i) уже обладают основным свойством. Процедура переставляет элементы поддерева с вершиной i, после чего оно обладает основным свойством. Идея проста: если это свойство не выполнено для вершины i, то её следует поменять с бо́льшим из её детей и т. д., пока элемент $A[i]$ не *"подгрузится"* до нужного места.

Рисунок 7.2 Работа процедуры Heapify$(A, 2)$ при *heap-size*$[A] = 10$. (а) Начальное состояние кучи. В вершине $i = 2$ основное свойство нарушено. Чтобы восстановить его, необходимо поменять $A[2]$ и $A[4]$. После этого (б) основное свойство нарушается в вершине с индексом 4. Рекурсивный вызов процедуры Heapify$(A, 4)$ восстанавливает основное свойство в вершине с индексом 4 путём перестановки $A[4] \leftrightarrow A[9]$ (в). После этого основное свойство выполнено для всех вершин, так что процедура Heapify$(A, 9)$ уже ничего не делает.

```
HEAPIFY($A, i$)
1  $l \leftarrow$ LEFT($i$)
2  $r \leftarrow$ RIGHT($i$)
3  if $l \leqslant heap\text{-}size[A]$ и $A[l] > A[i]$
4     then $largest \leftarrow l$
5     else $largest \leftarrow i$
6  if $r \leqslant heap\text{-}size[A]$ и $A[r] > A[largest]$
7     then $largest \leftarrow r$
8  if $largest \neq i$
9     then обменять $A[i] \leftrightarrow A[largest]$
10         HEAPIFY($A, largest$)
```

Работа процедуры Heapify показана на рис. 7.2. В строках 3–7 в переменную *largest* помещается индекс наибольшего из элементов $A[i]$, $A[\text{Left}(i)]$ и $A[\text{Right}(i)]$. Если $largest = i$, то элемент $A[i]$ уже *"погрузился"* до нужного места, и работа процедуры закончена. Иначе процедура меняет местами $A[i]$ и $A[largest]$ (что обеспечивает выполнение свойства (7.1) в вершине i, но возможно, нарушает это свойство в вершине *largest*) и рекурсивно вызывает себя для вершины *largest*, чтобы исправить возможные нарушения.

Оценим время работы процедуры Heapify. На каждом шаге требуется произвести $\Theta(1)$ действий, не считая рекурсивного вызова. Пусть $T(n)$ — время работы для поддерева, содержащего n элементов. Если поддерево с корнем i состоит из n элементов, то поддеревья с корнями Left(i) и Right(i) содержат не более чем по $2n/3$ элементов каждое (наихудший случай — когда последний уровень в поддереве заполнен наполовину). Таким образом,

$$T(n) \leqslant T(2n/3) + \Theta(1)$$

Из теоремы 4.1 (случай 2) получаем, что $T(n) = O(\lg n)$. Эту же оценку можно получить так: на каждом шаге мы спускаемся по дереву на один уровень, а высота дерева есть $O(\lg n)$.

Упражнения

7.2-1 Покажите, следуя образцу рис. 7.2, как работает процедура Heapify(A, 3) для массива $\langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0\rangle$.

7.2-2 Пусть элемент $A[i]$ больше, чем его дети. Каков будет результат вызова процедуры Heapify(A, i)?

7.2-3 Пусть $i > \textit{heap-size}[A]/2$. Каков будет результат вызова процедуры Heapify(A, i)?

7.2-4 Измените процедуру Heapify, заменив рекурсию циклом. (Некоторые компиляторы при этом порождают более эффективный код.)

7.2-5 Докажите, что наибольшее время работы процедуры Heapify для кучи из n элементов равно $\Omega(\lg n)$. (Указание: приведите пример, когда процедура вызывается для каждой вершины на пути от корня к листу).

7.3 Построение кучи

Пусть дан массив $A[1\,..\,n]$, который мы хотим превратить в кучу, переставив его элементы. Для этого можно использовать процедуру Heapify, применяя её по очереди ко всем вершинам, начиная с нижних. Поскольку вершины с номерами $\lfloor n/2\rfloor + 1, \ldots, n$ являются листьями, поддеревья с этими вершинами удовлетворяют основному свойству. Для каждой из оставшихся вершин, в порядке убывания индексов, мы применяем процедуру Heapify. Порядок обработки вершин гарантирует, что каждый раз условия вызова процедуры (выполнение основного свойства для поддеревьев) будут выполнены.

```
Build-Heap(A)
1  heap-size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3       do Heapify(A, i)
```

Пример работы процедуры Build-Heap показан на рис. 7.3.

Ясно, что время работы процедуры Build-Heap не превышает $O(n \lg n)$. Действительно, процедура Heapify вызывается $O(n)$ раз, а каждое её выполнение требует времени $O(\lg n)$. Однако эту оценку можно улучшить, чем мы сейчас и займёмся.

Дело в том, что время работы процедуры Heapify зависит от высоты вершины, для которой она вызывается (и пропорционально этой высоте). Поскольку число вершин высоты h в куче из n элементов не превышает $\lceil n/2^{h+1} \rceil$ (см. упр. 7.3-3), а высота все кучи не превышает $\lfloor \lg n \rfloor$ (упр. 7.1-2), время работы процедуры Build-Heap не превышает

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) \tag{7.2}$$

Полагая $x = 1/2$ в формуле (3.6), получаем верхнюю оценку для суммы в правой части:

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2.$$

Таким образом, время работы процедуры Build-Heap составляет

$$O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n).$$

Упражнения

7.3-1 Покажите, следуя образцу рис. 7.3, как работает процедура Build-Heap для массива $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

7.3-2 Почему в процедуре Build-Heap существенно, что параметр i пробегает значения от $\lfloor length[A]/2 \rfloor$ до 1 (а не наоборот)?

7.3-3 Докажите, что куча из n элементов содержит не более $\lceil n/2^{h+1} \rceil$ вершин высоты h.

Рисунок 7.3 Работа процедуры Build-Heap. Показано состояние данных перед каждым вызовом процедуры Heapify в строке 3.

7.4 Алгоритм сортировки с помощью кучи

Алгоритм сортировки с помощью кучи состоит из двух частей. Сначала вызывается процедура Build-Heap, после выполнения которой массив является кучей. Идея второй части проста: максимальный элемент массива теперь находится в корне дерева ($A[1]$). Его следует поменять с $A[n]$, уменьшить размер кучи на 1 и восстановить основное свойство в корневой вершине (поскольку поддеревья с корнями Left(1) и Right(1) не утратили основного свойства кучи, это можно сделать с помощью процедуры Heapify). После этого в корне будет находиться максимальный из оставшихся элементов. Так делается до тех пор, пока в куче не останется всего один элемент.

```
Heapsort($A$)
1  Build-Heap($A$)
2  for $i \leftarrow length[A]$ downto 2
3      do поменять $A[1] \leftrightarrow A[i]$
4         $heap\text{-}size[A] \leftarrow heap\text{-}size[A] - 1$
5         Heapify($A, 1$)
```

Работа второй части алгоритма показана на рис. 7.4. Изображены состояния кучи перед каждым выполнением цикла **for** (строка 2).

Время работы процедуры Heapsort составляет $O(n \lg n)$. Действительно, первая часть (построение кучи) требует времени $O(n)$, а каждое из $n-1$ выполнений цикла **for** занимает время $O(\lg n)$.

Упражнения

7.4-1 Покажите, следуя образцу рис. 7.4, как работает процедура Heapsort для массива $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

7.4-2 Пусть исходный массив A уже отсортирован в порядке возрастания. Каково будет время сортировки с помощью кучи? А если массив был отсортирован в порядке убывания?

7.4-3 Докажите, что время работы процедуры Heapsort составляет $\Omega(n \lg n)$.

7.5 Очереди с приоритетами

На практике алгоритм сортировки с помощью кучи не является самым быстрым — как правило, быстрая сортировка (гл. 8) работает быстрее. Однако сама куча как структура данных часто ока-

Рисунок 7.4 Работа процедуры Heapsort. Показано состояние массива перед каждым вызовом процедуры Heapify. Зачернённые элементы уже не входят в кучу.

зывается полезной. В этом разделе мы рассмотрим моделирование очереди с приоритетами на базе кучи — один из самых известных примеров использования кучи.

Очередь с приоритетами (priority queue) — это множество S, элементы которого мы будем считать числами. На практике элементами множества S являются пары $\langle key, \alpha\rangle$, где key — число, определяющее приоритет элемента и называемое **ключом** (key). а α — связанная с ним информация; эта информация хранится рядом с элементом и перемещается вместе с ним, не влияя на его обработку.

Возможны следующие операции над очередью с приоритетами:

Insert(S, x): добавление элемента x к множеству S;

Maximum(S): наибольший элемент множества;

Extract-Max(S): изъятие из множества наибольшего элемента.

Очередь с приоритетами может, например, использоваться в операционной системе с разделением времени. При этом хранится список заданий с приоритетами; как только выполнение очередного задания заканчивается, из очереди выбирается задание с наибольшим приоритетом (операция Extract-Max). Новые задания добавляются в очередь с помощью операции Insert.

Другое применение той же структуры — управляемое событиями моделирование (event-driven simulation). В очереди находятся события, а приоритет определяется временем, когда событие должно произойти. Разумеется, события должны моделироваться в том порядке, в котором они происходят. Выбор очередного события производится с помощью операции Extract-Min (порядок здесь обратный), добавление событий — с помощью операции Insert.

Опишем теперь реализацию очереди с приоритетами. Будем хранить элементы множества в виде кучи. При этом максимальный элемент находится в корне, так что операция Maximum требует времени $\Theta(1)$. Чтобы изъять максимальный элемент из очереди, нужно действовать так же, как и при сортировке:

```
Heap-Extract-Max($A$)
if $heap\text{-}size[A] < 1$
   then ошибка: "очередь пуста"
$max \leftarrow A[1]$
$A[1] \leftarrow A[heap\text{-}size[A]]$
$heap\text{-}size[A] \leftarrow heap\text{-}size[A] - 1$
Heapify($A, 1$)
return $max$
```

Время работы составляет $O(\lg n)$ (процедура Heapify вызывается один раз).

Чтобы добавить элемент к очереди, его следует добавить в конец кучи (как лист), а затем дать ему *"всплыть"* до нужного места:

```
Heap-Insert($A, key$)
$heap\text{-}size[A] \leftarrow heap\text{-}size[A] + 1$
$i \leftarrow heap\text{-}size[A]$
while $i > 1$ и $A[$Parent$(i)] < key$
    do $A[i] \leftarrow A[$Parent$(i)]$
       $i \leftarrow$ Parent$(i)$
$A[i] \leftarrow key$
```

Пример работы процедуры Heap-Insert показан на рис. 7.5. Время работы составляет $O(\lg n)$, поскольку *"подъём"* нового листа занимает не более $\lg n$ шагов (индекс i после каждой итерации цикла **while** уменьшается по крайней мере вдвое).

Итак, все операции над очередью с приоритетами из n элементов

Рисунок 7.5 Работа процедуры Heap-Insert. Добавляется элемент с ключевым значением 15 (темный кружок означает место для этого элемента).

требуют времени $O(\lg n)$.

Упражнения

7.5-1 Покажите, следуя образцу рис. 7.5, как работает процедура Heap-Insert$(A, 3)$ для кучи $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1\rangle$.

7.5-2 Покажите на рисунках, как работает процедура Heap-Extract-Max для кучи из предыдущего упражнения.

7.5-3 Объясните, как реализовать обычную очередь (first-in, first-out) и стек на базе очереди с приоритетами. (Определения см. в разд. 11.1).

7.5-4 Реализуйте процедуру Heap-Increase-Key(A, i, k) (увеличение элемента), которая увеличивает элемент $A[i]$ до k, если он был меньше k ($A[i] \leftarrow \max(A[i], k)$) и восстанавливает основное свойство кучи. Время работы — $O(\lg n)$.

7.5-5 Реализуйте операцию Heap-Delete(A, i) — удаление элемента с индексом i из кучи. Время работы $O(\lg n)$.

7.5-6 Придумайте алгоритм, который позволяет за время $O(n \lg k)$ слить k отсортированных списков в один отсортированный список (здесь n — общее число элементов в списках). (Указание: используйте кучу.)

Задачи

7-1 Построение кучи с помощью вставок

Можно построить кучу, последовательно добавляя элементы с помощью процедуры Heap-Insert. Рассмотрим следующий алгоритм:

```
Build-Heap'(A)
heap-size[A] ← 1
for i ← 2 to length[A]
    do Heap-Insert(A, A[i])
```

а. Запустим процедуры Build-Heap и Build-Heap′ для одного и того же массива. Всегда ли они создадут одинаковые кучи? (Докажите или приведите контрпример.)

б. Докажите, что время работы процедуры Build-Heap′ в худшем случае составляет $\Theta(n \lg n)$ (где n — количество элементов).

7-2 Работа с d-ичными кучами

Рассмотрим **d-ичную кучу** (d-ary heap), в которой вершины имеют d детей вместо двух.

а. Как выглядят для такой кучи процедуры, аналогичные Parent, Left и Right?

б. Как высота d-ичной кучи из n элементов выражается через n и d?

в. Реализуйте процедуру Extract-Max. Каково время её работы (выразите его через n и d)?

г. Реализуйте процедуру Insert. Каково время её работы?

д. Реализуйте процедуру Heap-Increase-Key (упр. 7.5-4). Каково время её работы?

Замечания

Алгоритм сортировки с помощью кучи предложил Уильямс [202]; там же описана реализация очереди с приоритетами на базе кучи. Процедура Build-Heap предложена Флойдом [69].

8 Быстрая сортировка

В этой главе рассмотрен так называемый алгоритм *"быстрой сортировки"*. Хотя время его работы для массива из n чисел в худшем случае составляет $\Theta(n^2)$, на практике этот алгоритм является одним из самых быстрых: математическое ожидание времени работы составляет $\Theta(n \lg n)$, причём множитель при $n \lg n$ довольно мал. Кроме того, быстрая сортировка не требует дополнительной памяти и сохраняет эффективность для систем с виртуальной памятью.

В разделе 8.1 описывается алгоритм в целом и процедура разделения массива на части. Оценка эффективности алгоритма довольно сложна; в разделе 8.2 приводятся интуитивные доводы, а строгий анализ отложен до раздела 8.4. В разделе 8.3 описаны варианты быстрой сортировки, использующие генератор случайных чисел. При этом время работы в худшем случае (при неудачном случайном выборе) составляет $O(n^2)$, но среднее время работы составляет лишь $O(n \lg n)$. (Говоря о среднем времени, мы имеем в виду не усреднение по всем входам, а математическое ожидание времени работы, которое *для любого входа* не превосходит $O(n \log n)$.) Один из вариантов вероятностного алгоритма быстрой сортировки подробно анализируется в разделе 8.4, где доказаны оценки для среднего и наибольшего времени работы.

8.1 Описание быстрой сортировки

Быстрая сортировка (quicksort), как и сортировка слиянием, основана на принципе *"разделяй и властвуй"* (см. разд. 1.3.1). Сортировка участка $A[p\,..\,r]$ происходит так:

- Элементы массива A переставляются так, чтобы любой из элементов $A[p], \ldots, A[q]$ был не больше любого из элементов $A[q+1], \ldots, A[r]$, где q — некоторое число в интервале $p \leqslant q < r$. Эту операцию мы будем называть разделением (partition).
- Процедура сортировки рекурсивно вызывается для массивов $A[p\,..\,q]$ и $A[q+1\,..\,r]$.

После этого массив $A[p\,..\,r]$ отсортирован.

Итак, процедура сортировки Quicksort выглядит следующим образом:

```
Quicksort($A, p, r$)
if $p < r$
   then $q \leftarrow$ Partition($A, p, r$)
        Quicksort($A, p, q$)
        Quicksort($A, q+1, r$)
```

Для сортировки всего массива необходимо выполнить процедуру Quicksort($A, 1, length[A]$).

Разбиение массива

Основной шаг алгоритма — процедура Partition, которая переставляет элементы массива $A[p\,..\,r]$ нужным образом:

```
Partition($A, p, r$)
 1  $x \leftarrow A[p]$
 2  $i \leftarrow p-1$
 3  $j \leftarrow r+1$
 4  while TRUE
 5      do repeat $j \leftarrow j-1$
 6           until $A[j] \leqslant x$
 7         repeat $i \leftarrow i+1$
 8           until $A[i] \geqslant x$
 9         if $i < j$
10           then поменять $A[i] \leftrightarrow A[j]$
11           else return $j$
```

Работа процедуры Partition показана на рис. 8.1. Элемент $x = A[p]$ выбирается в качестве *"граничного"*; массив $A[p\,..\,q]$ будет содержать элементы, не большие x, а массив $A[q+1\,..\,r]$ — элементы, не меньшие x. Идея состоит в том, чтобы накапливать элементы, не большие x, в начальном отрезке массива ($A[p\,..\,i]$), а элементы, не меньшие x — в конце ($A[j\,..\,r]$). В начале оба *"накопителя"* пусты: $i = p-1$, $j = r+1$.

Внутри цикла **while** (в строках 5–8) к начальному и конечному участкам присоединяются элементы (как минимум по одному). После выполнения этих строк $A[i] \geqslant x \geqslant A[j]$. Если мы поменяем $A[i]$ и $A[j]$ местами, то их можно будет присоединить к начальному и конечному участкам.

В момент выхода из цикла выполнено неравенство $i \geqslant j$. При этом массив разбит на части $A[p], \ldots, A[j]$ и $A[j+1], \ldots, A[r]$; любой элемент первой части не превосходит любого элемента второй

Рисунок 8.1 Работа процедуры Partition. Незакрашенные элементы относятся к уже сформированным кускам, закрашенные ещё не распределены. (а) Начальное состояние массива, начальный и конечный куски пусты. Элемент $x = A[p] = 5$ используется в качестве граничного. (б) Результат первого прохода цикла **while** (строки 4–8). (в) Элементы $A[i]$ и $A[j]$ меняются местами (строка 10). (г) Результат второго прохода цикла **while**. (д) Результат третьего (последнего) прохода цикла **while**. Поскольку $i \geqslant j$, процедура останавливается и возвращает значение $q = j$. Элементы слева от $A[j]$ (включая сам этот элемент) не больше, чем $x = 5$, а элементы справа от $A[j]$ не меньше, чем $x = 5$.

части. Процедура возвращает значение j.

Хотя идея процедуры очень проста, сам алгоритм содержит ряд тонких моментов. Например, не очевидно, что индексы i и j не выходят за границы промежутка $[p\,..\,r]$ в процессе работы. Другой пример: важно, что в качестве граничного значения выбирается $A[p]$, а не, скажем, $A[r]$. В последнем случае может оказаться, что $A[r]$ — самый большой элемент массива, и в конце выполнения процедуры будет $i = j = r$, так что возвращать $q = j$ будет нельзя — нарушится требование $q < r$, и процедура Quicksort зациклится. Правильность процедуры Partition составляет предмет задачи 8-1.

Время работы процедуры Partition составляет $\Theta(n)$, где $n = r - p + 1$ (см. упр. 8.1-3).

Упражнения

8.1-1 Покажите, следуя образцу рис. 8.1, как работает процедура Partition для массива $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$.

8.1-2 Пусть все элементы массива $A[p\,..\,r]$ равны. Какое значение вернёт процедура Partition?

8.1-3 Приведите простое соображение, объясняющее, почему время работы процедуры Partition составляет $\Theta(n)$.

8.1-4 Как отсортировать массив в порядке убывания (а не возрастания), используя те же методы?

8.2 Работа быстрой сортировки

Время работы алгоритма быстрой сортировки зависит от того, как разбивается массив на каждом шаге. Если разбиение происходит на примерно равные части, время работы составляет $O(n \lg n)$, как и для сортировки слиянием. Если же размеры частей сильно отличаются, сортировка может занимать время $O(n^2)$, как при сортировке вставками.

Наихудшее разбиение

"*Наиболее неравные части*" получатся, если одна часть содержит $n-1$ элемент, а вторая — всего 1. (Как мы увидим в разделе 8.4.1, это наихудший случай с точки зрения времени работы.) Предположим, что на каждом шаге происходит именно так. Поскольку процедура разбиения занимает время $\Theta(n)$, для времени работы $T(n)$ получаем соотношение

$$T(n) = T(n-1) + \Theta(n)$$

Поскольку $T(1) = \Theta(1)$, имеем

$$T(n) = T(n-1) + \Theta(n) = \sum_{k=1}^{n} \Theta(k) = \Theta\left(\sum_{k=1}^{n} k\right) = \Theta(n^2)$$

(последняя сумма — арифметическая прогрессия, см. формулу (3.2)). Дерево рекурсии для этого случая показано на рис. 8.2. (По поводу деревьев рекурсии см. разд. 4.2.)

Мы видим, что при максимально несбалансированном разбиении время работы составляет $\Theta(n^2)$, как и для сортировки вставками. В частности, это происходит, если массив изначально отсортирован (заметим, что в этом случае сортировка вставками производится за время $\Theta(n)$).

Наилучшее разбиение

Если на каждом шаге массив разбивается ровно пополам, быстрая сортировка требует значительно меньше времени. Действительно, в этом случае рекуррентное соотношение имеет вид

$$T(n) = 2T(n/2) + \Theta(n)$$

и, согласно теореме 4.1 (случай 2), $T(n) = \Theta(n \lg n)$. Дерево рекурсии для этого случая показано на рисунке 8.3.

Рисунок 8.2 Дерево рекурсии процедуры Quicksort для наихудшего случая (процедура Partition каждый раз производит разбиение, в котором одна из частей содержит один элемент). Время работы равно $\Theta(n^2)$.

Рисунок 8.3 Дерево рекурсии процедуры Quicksort для наилучшего случая (массив каждый раз разбивается пополам). Время работы равно $\Theta(n \lg n)$.

Промежуточный случай

Как будет показано в разделе 8.4, среднее время работы (с точностью до множителя) совпадает с временем работы в наилучшем случае. Чтобы объяснить это, посмотрим, как меняется рекуррентное соотношение в зависимости от степени сбалансированности разбиения.

Пусть, например, на каждом шаге массив разбивается на две части с отношением размеров $9:1$. Тогда

$$T(n) = T(9n/10) + T(n/10) + n$$

(для удобства мы заменили $\Theta(n)$ на n). Дерево рекурсии показано на рисунке 8.4. На каждом уровне мы производим не более n действий, так что время работы определяется глубиной рекурсии.

Рисунок 8.4 Дерево рекурсии для случая, когда разбиение каждый раз производится в отношении $9:1$. Время работы равно $\Theta(n \lg n)$.

В данном случае эта глубина равна $\log_{10/9} n = \Theta(\lg n)$, так что время работы по-прежнему составляет $\Theta(n \lg n)$, хотя константа и больше. Ясно, что для любого фиксированного отношения размеров частей (сколь бы велико оно ни было) глубина дерева рекурсии по-прежнему будет логарифмической, а время работы будет равно $\Theta(n \lg n)$.

Среднее время: интуитивные соображения

Чтобы вопрос о среднем времени работы имел смысл, нужно уточнить, с какой частотой появляются различные входные значения. Как правило, предполагается, что все перестановки входных значений равновероятны. (Мы вернёмся к этому в следующем разделе.)

Для наугад взятого массива разбиения вряд ли будут всё время происходить в одном и том же отношении — скорее всего, часть разбиений будет хорошо сбалансирована, а часть нет. Как показывает упр. 8.2-5, примерно 80 процентов разбиений производятся в отношении не более $9:1$.

Будем предполагать для простоты, что на каждом втором уровне все разбиения наихудшие, а на оставшейся половине уровней наилучшие (пример показан на рис. 8.5(a)). Поскольку после каждого ”*хорошего*” разбиения размер частей уменьшается вдвое, число ”*хороших*” уровней равно $\Theta(\lg n)$, а поскольку каждый второй уровень ”*хороший*”, общее число уровней равно $\Theta(\lg n)$, а время работы — $\Theta(n \lg n)$. Таким образом, плохие уровни не испортили асимптотику времени работы (а лишь увеличили константу, скрытую в асимпто-

Рисунок 8.5 (а) Два уровня: плохой (n разбивается на $n-1$ и 1) и хороший ($n-1$ разбивается на две равные части). (б) Если эти два уровня заменить одним, получится разбиение на почти равные по величине части.

тическом обозначении).

Упражнения

8.2-1 Докажите, что если массив состоит из одинаковых элементов, то время работы процедуры QUICKSORT равно $\Theta(n \lg n)$.

8.2-2 Пусть массив отсортирован в порядке убывания. Докажите, что время работы процедуры QUICKSORT составляет $\Theta(n^2)$.

8.2-3 Как правило, в банках обрабатывают чеки в порядке их поступления; клиенты же предпочитают, чтобы в отчёте платежи были указаны в порядке номеров чеков. Владелец чековой книжки обычно выписывает чеки подряд, а получатели чеков предъявляют их в банк вскоре после выписывания. Таким образом, порядок номеров нарушается незначительно. Следовательно, банку требуется отсортировать почти отсортированный массив. Объясните, почему сортировка вставками в таких случаях работает быстрее, чем быстрая сортировка.

8.2-4 Пусть разбиения на каждом шаге производятся в отношении $\alpha : 1-\alpha$, где $0 < \alpha \leqslant 1/2$. Докажите, что минимальная глубина листа на дереве рекурсии примерно равна $-\lg n / \lg \alpha$, а максимальная примерно равна $-\lg n / \lg(1-\alpha)$. (Не заботьтесь об округлении.)

8.2-5* Докажите, что для любого числа α в интервале $0 < \alpha \leqslant 1/2$ вероятность того, что разбиение случайного массива будет сбалансировано не хуже, чем $\alpha : 1 - \alpha$, примерно равна $1 - 2\alpha$. При каком значении α вероятность этого события равна $1/2$?

8.3 Вероятностные алгоритмы быстрой сортировки

Ранее мы предположили, что все перестановки входных значений равновероятны. Если это так, а размер массива достаточно велик, быстрая сортировка — один из наиболее эффективных алгоритмов. На практике, однако, это предположение (равной вероятности всех перестановок на входе) не всегда оправдано (см. упр. 8.2-3). В этом разделе мы введём понятие вероятностного алгоритма и рассмотрим два вероятностных алгоритма быстрой сортировки, которые позволяют отказаться от предположения о равной вероятности всех перестановок.

Идея состоит в привнесении случайности, обеспечивающем нужное распределение. Например, перед началом сортировки можно случайно переставить элементы, после чего уже все перестановки станут равновероятными (это можно сделать за время $O(n)$ — см. упр. 8.3-4). Такая модификация не увеличивает существенно время работы, но теперь математическое ожидание времени работы не зависит от порядка элементов во входном массиве (они всё равно случайно переставляются).

Алгоритм называется **вероятностным** (randomized), если он использует **генератор случайных чисел** (random-number generator). Мы будем считать, что генератор случайных чисел RANDOM работает так: RANDOM(a, b) возвращает с равной вероятностью любое целое число в интервале от a до b. Например, RANDOM$(0, 1)$ возвращает 0 или 1 с вероятностью $1/2$. При этом разные вызовы процедуры независимы в смысле теории вероятностей. Можно считать, что мы каждый раз бросаем кость с $(b - a + 1)$ гранями и сообщаем номер выпавшей грани. (На практике обычно используют **генератор псевдослучайных чисел** (pseudorandom-number generator) — детерминированный алгоритм, который выдаёт числа, *"похожие"* на случайные.)

Для такого вероятностного варианта алгоритма варианта быстрой сортировки нет *"неудобных входов"*: упражнение 13.4-4 показывает, что перестановок, при которых время работы велико, совсем мало — поэтому вероятность того, что алгоритм будет работать долго (для любого конкретного входа) невелика.

Аналогичный подход применим и в других ситуациях, когда в ходе выполнения алгоритма мы должны выбрать один из многих вариантов его продолжения, причём мы не знаем, какие из них хорошие, а какие плохие, но знаем, что хороших вариантов достаточно много. Нужно только, чтобы плохие выборы не разрушали достигнутого при предыдущих хороших (как мы видели в разделе 8.2, для алгоритма быстрой сортировки это так).

Вместо того, чтобы предварительно переставлять элементы массива, мы можем внести элемент случайности в процедуру Partition. Именно, перед разбиением массива $A[p\,..\,r]$ будем менять элемент $A[p]$ со случайно выбранным элементом массива. Тогда каждый элемент с равной вероятностью может оказаться граничным, и в среднем разбиения будут получаться достаточно сбалансированными.

Этот подход заменяет разовую случайную перестановку входов в начале использованием случайных выборов на всём протяжении работы алгоритма. В сущности это то же самое, и оба алгоритма имеют математическое ожидание времени работы $O(n \lg n)$, но небольшие технические различия делают анализ нового варианта проще, и именно он будет рассматриваться в разделе 8.4.

Изменения, которые нужно внести в процедуры, совсем невелики:

```
Randomized-Partition($A, p, r$)
$i \leftarrow$ Random($p, r$)
поменять $A[p] \leftrightarrow A[i]$
return Partition($A, p, r$)
```

В основной процедуре теперь будет использоваться Randomized-Partition вместо Partition:

```
Randomized-Quicksort($A, p, r$)
if $p < r$
   then $q \leftarrow$ Randomized-Partition($A, p, r$)
        Randomized-Quicksort($A, p, q$)
        Randomized-Quicksort($A, q + 1, r$)
```

Анализом этого алгоритма мы займёмся в следующем разделе.

Упражнения

8.3-1 Почему для вероятностного алгоритма важно не максимальное время работы (для данного входа), а математическое ожидание этого времени?

8.3-2 Сколько раз при выполнении процедуры Randomized-Quicksort может происходить обращение к генератору случайных чисел Random в худшем случае? Изменится ли ответ для наилучшего случая?

8.3-3* Реализуйте процедуру Random(a, b), используя бросания монеты, т.е. датчик, с равной вероятностью выдающий 0 или 1. Каково математическое ожидание времени работы вашей процедуры?

8.3-4* Придумайте вероятностную процедуру, которая за время $\Theta(n)$ случайным образом переставляет элементы входного массива $A[1\,..\,n]$.

8.4 Анализ быстрой сортировки

В этом разделе мы превратим "*интуитивные*" соображения раздела 8.2 в строгое рассуждение. Сначала мы рассмотрим наихудший случай (рассуждения будут одинаковы и для алгоритма Quicksort, и для алгоритма Randomized-Quicksort), а затем найдём среднее время работы алгоритма Randomized-Quicksort.

8.4.1 Анализ наихудшего случая

В разделе 8.2 мы видели, что если разбиение на каждом шаге наиболее несбалансировано, то время работы составляет $\Theta(n^2)$. Интуитивно ясно, что это наихудший (в смысле времени работы) случай. Сейчас мы строго докажем это.

Для доказательства того, что время работы составляет $O(n^2)$, мы используем метод подстановки (см. разд. 4.1). Пусть $T(n)$ — наибольшее время работы алгоритма для массива длины n. Тогда, очевидно,

$$T(n) = \max_{1\leqslant q\leqslant n-1}(T(q) + T(n-q)) + \Theta(n) \qquad (8.1)$$

(мы рассматриваем все возможные разбиения на первом шаге). Предположим, что $T(q) \leqslant cq^2$ для некоторой константы c и для всех q, меньших некоторого n. Тогда

$$T(n) \leqslant \max_{1\leqslant q\leqslant n-1}(cq^2+c(n-q)^2)+\Theta(n) = c\cdot \max_{1\leqslant q\leqslant n-1}(q^2+(n-q)^2)+\Theta(n).$$

Квадратный трёхчлен $q^2 + (n-q)^2$ достигает максимума на отрезке $1 \leqslant q \leqslant n-1$ в его концах (вторая производная по q положительна, поэтому функция выпукла вниз, см. упр. 8.4-2). Этот максимум равен $1^2 + (n-1)^2 = n^2 - 2(n-1)$. Отсюда получаем

$$T(n) \leqslant cn^2 - 2c(n-1) + \Theta(n) \leqslant cn^2,$$

если константа c выбрана так, чтобы последнее слагаемое было меньше предпоследнего. Итак, время работы в худшем случае составляет $\Theta(n^2)$.

8.4.2 Анализ среднего времени работы

Как мы уже видели в разделе 8.2, если разбиения производятся так, что отношение размеров частей ограничено, то глубина дерева рекурсии равна $\Theta(\lg n)$, а время работы — $\Theta(n \lg n)$. Чтобы получить оценку среднего времени работы алгоритма Randomized-Quicksort, мы сначала проанализируем работу процедуры Partition, затем получим рекуррентное соотношение на среднее время работы и решим его (попутно получив одну полезную оценку).

Анализ разбиений

Напомним, что перед тем как в строке 3 процедуры Randomized-Partition вызывается процедура Partition, элемент $A[p]$ переставляется со случайно выбранным элементом массива $A[p\,..\,r]$. Для простоты мы будем предполагать, что все числа в массиве различны. Хотя оценка среднего времени сохраняется и в том случае, когда в массиве есть одинаковые элементы, получить её сложнее, и мы этого делать не будем.

Прежде всего, заметим, что значение q, которое возвратит процедура Partition, зависит только от того, сколько в массиве элементов, не больших $x = A[p]$ (число таких элементов мы будем называть **рангом** (rank) элемента x и обозначать $\mathrm{rank}(x)$). Если $n = r - p + 1$ — число элементов в массиве, то, поскольку все элементы имеют равные шансы попасть на место $A[p]$, все значения $\mathrm{rank}(x)$, от 1 до n, равновероятны (имеют вероятность $1/n$).

Если $\mathrm{rank}(x) > 1$, то, как легко видеть, при разбиении левая часть будет содержать $\mathrm{rank}(x) - 1$ элементов — в ней окажутся все элементы, меньшие x. Если же $\mathrm{rank}(x) = 1$, то левая часть будет содержать один элемент (после первого же выполнения цикла будет $i = j = p$). Отсюда следует, что с вероятностью $1/n$ левая часть будет содержать $2, 3, \ldots, n-1$ элементов, а с вероятностью $2/n$ — один элемент.

Рекуррентное соотношение для среднего времени работы

Обозначим среднее время работы алгоритма Randomized-Quicksort для массива из n элементов через $T(n)$. Ясно, что $T(1) = \Theta(1)$. Время работы состоит из времени работы процедуры Partition, которое составляет $\Theta(n)$, и времени работы для двух массивов размера q и $n-q$, причём q с вероятностью $2/n$ принимает значение 1 и с вероятностью $1/n$ — значения $2, \ldots, n-1$. Поэтому

$$T(n) = \frac{1}{n}\left(T(1) + T(n-1) + \sum_{q=1}^{n-1}(T(q) + T(n-q))\right) + \Theta(n) \quad (8.2)$$

(слагаемое, соответствующее $q = 1$, входит дважды). Поскольку $T(1) = \Theta(1)$ и $T(n) = O(n^2)$, имеем

$$\frac{1}{n}(T(1) + T(n-1)) = \frac{1}{n}(\Theta(1) + O(n^2)) = O(n).$$

Поэтому слагаемые $T(1)$ и $T(n-1)$ в первой скобке (8.2) можно включить в $\Theta(n)$. С учётом этого получаем

$$T(n) = \frac{1}{n}\sum_{q=1}^{n-1}(T(q) + T(n-q)) + \Theta(n). \quad (8.3)$$

Поскольку каждое слагаемое $T(k)$, где $k = 1, \ldots, n-1$, встречается в сумме дважды, её можно переписать так:

$$T(n) = \frac{2}{n}\sum_{k=1}^{n-1}T(k) + \Theta(n). \quad (8.4)$$

Решение рекуррентного соотношения

Соотношение (8.4) можно решить, используя метод подстановки. Предположим, что $T(n) \leqslant an\lg n + b$, где константы $a > 0$ и $b > 0$ пока неизвестны, и попытаемся доказать это по индукции. При $n = 1$ это верно, если взять достаточно большие a и b. При $n > 1$ имеем

$$\begin{aligned} T(n) &= \frac{2}{n}\sum_{k=1}^{n-1}T(k) + \Theta(n) \\ &\leqslant \frac{2}{n}\sum_{k=1}^{n-1}(ak\lg k + b) + \Theta(n) \\ &= \frac{2a}{n}\sum_{k=1}^{n-1}k\lg k + \frac{2b}{n}(n-1) + \Theta(n). \end{aligned}$$

Ниже мы покажем, что первую сумму можно оценить так:

$$\sum_{k=1}^{n-1}k\lg k \leqslant \frac{1}{2}n^2\lg n - \frac{1}{8}n^2. \quad (8.5)$$

Используя это, получим

$$\begin{aligned}T(n) &\leqslant \frac{2a}{n}\left(\frac{1}{2}n^2\lg n - \frac{1}{8}n^2\right) + \frac{2b}{n}(n-1) + \Theta(n)\\ &\leqslant an\lg n - \frac{a}{4}n + 2b + \Theta(n)\\ &= an\lg n + b + \left(\Theta(n) + b - \frac{a}{4}n\right) \leqslant an\lg n + b,\end{aligned}$$

если выбрать a так, чтобы $\frac{a}{4}n$ было больше $\Theta(n)+b$. Следовательно, среднее время работы есть $O(n\lg n)$.

Доказательство оценки для суммы

Осталось доказать оценку (8.5). Поскольку каждое слагаемое не превышает $n\lg n$, получаем оценку

$$\sum_{k=1}^{n-1} k\lg k \leqslant n^2\lg n.$$

Для наших целей она не подходит — нам необходима более точная оценка $\frac{1}{2}n^2\lg n - \Omega(n^2)$.

Если в предыдущей оценке заменять лишь $\lg k$ на $\lg n$, оставив k в неприкосновенности, получим оценку

$$\sum_{k=1}^{n-1} k\lg k \leqslant \lg n \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}\lg n \leqslant \frac{1}{2}n^2\lg n.$$

Осталось лишь заметить, что заменяя $\lg k$ на $\lg n$, мы прибавили по крайней мере по $k \cdot 1$ к каждому слагаемом первой половины суммы (где $k \leqslant n/2$), всего примерно $(n/2)^2/2 = n^2/8$.

Более формально,

$$\sum_{k=1}^{n-1} k\lg k = \sum_{k=1}^{\lceil n/2\rceil-1} k\lg k + \sum_{k=\lceil n/2\rceil}^{n-1} k\lg k$$

При $k < \lceil n/2\rceil$ имеем $\lg k \leqslant \lg(n/2) = \lg n - 1$. Поэтому

$$\begin{aligned}\sum_{k=1}^{n-1} k\lg k &\leqslant (\lg n - 1)\sum_{k=1}^{\lceil n/2\rceil-1} k + \lg n \sum_{k=\lceil n/2\rceil}^{n-1} k\\ &= \lg n\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2\rceil-1} k \leqslant \frac{1}{2}n(n-1)\lg n - \frac{1}{2}\left(\frac{n}{2}-1\right)\frac{n}{2}\\ &\leqslant \frac{1}{2}n^2\lg n - \frac{1}{8}n^2\end{aligned}$$

при $n \geqslant 2$. Оценка (8.5) доказана.

[Следующий простой вывод оценки для среднего времени работы вероятностного алгоритма быстрой сортировки (для несколько другого варианта алгоритма) предложил Л.А. Левин.

(1) Будем представлять себе сортировку так: есть N камней разного веса и чашечные весы для их сравнения. Мы берём случайный камень и делим всю кучу на три части: легче его, тяжелее его и он сам, после чего (рекурсивный вызов) сортируем первую и вторую части.

(2) Как выбрать случайный камень? Можно считать, что сначала всем камням случайно присваиваются различные ранги (будем считать их числами от 1 до N), и в качестве границы берётся камень минимального ранга (из подлежащих сортировке в данный момент). (Можно проверить, что это равносильно независимым выборам камней на каждом шаге: на первом шаге каждый из камней может быть выбран с равной вероятностью, после такого выбора в каждой из групп все камни также равновероятны и т.д.)

(3) Таким образом, каждый камень характеризуется двумя числами от 1 до N — порядковым *номером* (в порядке возрастания весов) и *рангом*. Соответствие между номерами и рангами определяет число операций в процессе сортировки.

(4) Для каждых двух номеров i, j из $\{1,\ldots,N\}$ через $p(i,j)$ обозначим вероятность того, что камни с этими номерами будут сравниваться друг с другом. Например, $p(i,i+1) = 1$, поскольку соседние по весу камни должны быть сравнены обязательно (сравнения с другими камнями их не различают).

(5) Заметим, что $p(i,i+2) = 2/3$. В самом деле, сравнения не произойдёт в том и только том случае, когда из трёх камней с номерами i, $i+1$ и $i+2$ камень $i+1$ имеет наименьший ранг. Аналогично, $p(i,i+k) = 2/(k+1)$ (камни с номерами i и $i+k$ сравниваются, если среди $k+1$ камней $i, i+1, \ldots, i+k$ один из двух крайних имеет наименьший ранг).

(6) Математическое ожидание числа сравнений можно разбить в сумму $\sum m(i,j)$ ожиданий числа сравнений между камнями с номерами i и j. Но поскольку два данных камня сравниваются не более одного раза, $m(i,j) = p(i,j)$. Таким образом, получаем точное выражение для математического ожидания числа сравнений:

$$\sum_{1\leqslant i<j\leqslant n} \frac{2}{j-i+1}.$$

(7) Группируя в этой сумме равные члены и вспоминая, что $1 + 1/2 + 1/3 + \ldots + 1/k = O(\lg k)$, получаем оценку $O(N \lg N)$ для математического ожидания времени работы быстрой сортировки.

Анализ облегчается тем, что алгоритм разбиения (на три части) более симметричен. Реализация такого способа разбиения также проста: массив делится на четыре участка (перечисляем их слева направо): меньшие границы, равные границе, непросмотренные и бóльшие границы.]

Упражнения

8.4-1 Докажите, что наименьшее время работы быстрой сортировки составляет $\Omega(n \lg n)$.

8.4-2 Докажите, что функция $q^2 + (n - q)^2$ на отрезке $[1, n - 1]$ принимает наибольшее значение в концах отрезка.

8.4-3 Докажите, что математическое ожидание времени работы процедуры Randomized-Quicksort на любом входе есть $\Omega(n \lg n)$.

8.4-4 На практике время работы быстрой сортировки можно уменьшить, если на завершающем этапе (когда массив почти отсортирован) использовать сортировку вставками. Сделать это можно, например, так: пусть процедура Randomized-Quicksort(A, p, r) ничего не делает, если $r - p + 1 < k$ (т.е. сортируемый массив содержит меньше k элементов). После окончания рекурсивных вызовов получившийся массив сортируется с помощью сортировки вставками. Докажите, что математическое ожидание времени работы такого алгоритма составляет $O(nk + n \lg(n/k))$. Как бы вы стали выбирать число k?

8.4-5* Докажите равенство

$$\int x \ln x \, dx = \frac{1}{2} x^2 \ln x - \frac{1}{4} x^2$$

Выведите отсюда (используя метод сравнения с интегралом) более сильную (в сравнении с (8.5)) оценку для суммы $\sum_{k=1}^{n-1} k \lg k$.

8.4-6* Рассмотрим следующую модификацию процедуры Randomized-Partition: случайным образом выбираются три элемента массива и в качестве граничного элемента берётся средний по величине из выбранных трёх камней. Оцените вероятность того, что при этом разбиение будет сбалансировано не хуже, чем $\alpha : 1 - \alpha$.

Задачи

8-1 Правильность процедуры разбиения

Покажите, что процедура Partition работает правильно. Для этого докажите следующее:

а. В процессе работы процедуры индексы i и j не выходят за пределы отрезка $[p\,..\,r]$.

б. В момент окончания работы процедуры индекс j не может быть равен r (т.е. обе части разбиения непусты).

в. В момент окончания работы процедуры любой элемент массива $A[p\,..\,j]$ не больше любого элемента массива $A[j+1\,..\,r]$.

8-2 Алгоритм Ломуто для разбиения

Вариант процедуры PARTITION, который мы сейчас рассмотрим, принадлежит Н. Ломуто (N. Lomuto). В процессе работы строятся куски $A[p\,..\,i]$ и $A[i+1\,..\,j]$, причём элементы первого куска не больше $x = A[r]$, а элементы второго куска — больше x.

```
LOMUTO-PARTITION($A, p, r$)
$x \leftarrow A[r]$
$i \leftarrow p - 1$
for $j \leftarrow p$ to $r$
    do if $A[j] \leqslant x$
          then $i \leftarrow i + 1$
               поменять $A[i] \leftrightarrow A[j]$
if $i < r$
  then return $i$
  else return $i - 1$
```

а. Докажите, что процедура LOMUTO-PARTITION работает правильно.

б. Сколько раз процедуры PARTITION и LOMUTO-PARTITION могут перемещать один и тот же элемент? (Укажите наибольшие значения.)

в. Докажите, что процедура LOMUTO-PARTITION, как и процедура PARTITION, требует времени $\Theta(n)$, где n — число элементов в массиве.

г. Заменим в тексте процедуры QUICKSORT процедуру PARTITION на LOMUTO-PARTITION. Как изменится время быстрой сортировки для массива, все элементы которого равны?

д. Рассмотрим процедуру RANDOMIZED-LOMUTO-PARTITION, которая меняет $A[r]$ со случайно выбранным элементом массива и затем вызывает процедуру LOMUTO-PARTITION. Докажите, что вероятность того, что процедура RANDOMIZED-LOMUTO-PARTITION вернёт значение q, равна вероятности того, что процедура RANDOMIZED-PARTITION вернёт значение $p + r - q$.

8-3 Сортировка по частям

Профессор предложил следующий *"продвинутый"* алгоритм сортировки:

```
STOOGE-SORT($A, i, j$)
if $A[i] > A[j]$
    then поменять $A[i] \leftrightarrow A[j]$
if $i + 1 \geqslant j$
    then return
$k \leftarrow \lfloor (j - i + 1)/3 \rfloor$             ▷ Округление с недостатком.
STOOGE-SORT($A, i, j - k$)   ▷ Первые две трети.
STOOGE-SORT($A, i + k, j$)   ▷ Последние две трети.
STOOGE-SORT($A, i, j - k$)   ▷ Опять первые две трети.
```

а. Докажите, что процедура STOOGE-SORT действительно сортирует массив.

б. Найдите рекуррентное соотношение для наибольшего времени работы процедуры STOOGE-SORT и получите из него оценку этого времени.

в. Сравните наибольшее время работы процедуры STOOGE-SORT с наибольшим временем для других вариантов сортировки (вставками, слиянием, с помощью кучи и быстрой сортировки). Стоит ли продлевать контракт с профессором?

8-4 Размер стека при быстрой сортировке

Процедура QUICKSORT два раза рекурсивно вызывает себя (для левой и для правой части). В действительности без второго рекурсивного вызова можно обойтись, заменив его циклом (именно так хорошие компиляторы обрабатывают ситуацию, когда последним оператором процедуры является рекурсивный вызов; для такой ситуации есть термин **tail recursion**):

```
QUICKSORT′($A, p, r$)
while $p < r$
    do ▷ Разбить и отсортировать левую часть.
        $q \leftarrow$ PARTITION($A, p, r$)
        QUICKSORT′($A, p, q$)
        $p \leftarrow q + 1$
```

а. Докажите, что процедура QUICKSORT′ действительно сортирует массив.

Как правило, компиляторы реализуют рекурсию с помощью стека, где хранятся копии локальных переменных для каждого рекурсивного вызова. Вершина стека содержит информацию, относящуюся к текущему вызову; когда он завершается, информация удаляется из стека. В нашем случае для каждого рекурсивного вызова локальные переменные занимают объём $O(1)$, так что необходимый **размер стека** (stack depth) пропорционален глубине рекурсии.

б. Покажите, что в некоторых случаях процедура QUICKSORT′ требует стека размера $\Theta(n)$.

в. Измените процедуру QUICKSORT$'$ так, чтобы объём стека не превышал $\Theta(\lg n)$ (сохраняя оценку $\Theta(n \lg n)$ для среднего времени работы).

8-5 Разбиение с помощью медианы трёх элементов

Работу процедуры RANDOMIZED-QUICKSORT можно ускорить, выбирая граничный элемент для разбиения более тщательно. Один из распространённых подходов — это **метод медианы трёх** (median-of-3 method): в качестве граничного элемента используется средний из трёх случайно выбранных элементов массива. Мы будем предполагать, что все элементы входного массива $A[1\,..\,n]$ различны и $n \geqslant 3$. Через $A'[1\,..\,n]$ будем обозначать отсортированный массив (который мы и хотим получить). Пусть $p_i = \mathsf{P}\{x = A'[i]\}$, где x — граничный элемент, выбранный описанным выше способом.

а. Выразите вероятность p_i (для $i = 2, 3, \ldots, n-1$) через i и n (заметьте, что $p_1 = p_n = 0$).

б. Насколько вероятность выбрать средний элемент ($A'[\lfloor (n+1)/2 \rfloor]$) больше, чем при обычном случайном выборе? К чему стремится отношение этих вероятностей при $n \to \infty$?

в. Будем называть разбиение с граничным элементом x *"хорошим"*, если $x = A'[i]$, где $n/3 \leqslant i \leqslant 2n/3$. Насколько вероятность *"хорошего"* разбиения больше, чем при обычном случайном выборе? (Указание: при вычислениях оцените сумму интегралом.)

г. Докажите, что использование метода медианы трёх сохраняет оценку $\Omega(n \lg n)$ для времени работы быстрой сортировки (и влияет лишь на константу перед $n \lg n$).

Замечания

Алгоритм быстрой сортировки принадлежит Хоару [98]. В статье Седжвика [174] обсуждаются детали реализации быстрой сортировки и их влияние на время работы. Вероятностные алгоритмы для разных задач рассматриваются в статье Рабина [165].

9 Сортировка за линейное время

Мы познакомились с различными алгоритмами, которые могут отсортировать n чисел за время $O(n \lg n)$. Алгоритмы сортировки слиянием (merge sort) и сортировки с помощью кучи (heap sort) работают за такое время в худшем случае, а у алгоритма быстрой сортировки таковым является среднее время работы. Оценка $O(n \lg n)$ точна: для каждого из этих алгоритмов можно предъявить последовательность из n чисел, время обработки которой будет $\Omega(n \lg n)$.

Все упомянутые алгоритмы проводят сортировку, основываясь исключительно на попарных сравнениях элементов, поэтому их иногда называют **сортировками сравнением** (comparison sort). В разделе 9.1 мы покажем, что всякий алгоритм такого типа сортирует n элементов за время не меньше $\Omega(n \lg n)$ в худшем случае. Тем самым алгоритмы сортировки слиянием и с помощью кучи асимптотически оптимальны: не существует алгоритма сортировки сравнением, который превосходил бы указанные алгоритмы более, чем в конечное число раз.

В разделах 9.2, 9.3 и 9.4 мы рассматриваем три алгоритма сортировки (сортировка подсчётом, цифровая сортировка и сортировка вычёрпыванием), работающих за линейное время. Разумеется, они улучшают оценку $\Omega(n \lg n)$ за счёт того, что используют не только попарные сравнения, но и внутреннюю структуру сортируемых объектов.

9.1 Нижние оценки для сортировки

Говорят, что алгоритм сортировки основан на сравнениях, если он никак не использует внутреннюю структуру сортируемых элементов, а лишь сравнивает их и после некоторого числа сравнений выдаёт ответ (указывающий истинный порядок элементов). Так мы приходим к модели алгоритмов сортировки, называемой разрешающими деревьями.

Рисунок 9.1 Разрешающее дерево для алгоритма сортировки вставками, обрабатывающего последовательность из трёх элементов. Поскольку число перестановок из трёх элементов равно $3! = 6$, у дерева должно быть не менее 6 листьев.

Разрешающие деревья

Начнём с примера: на рис. 9.1 изображено **разрешающее дерево** (decision tree), соответствующее сортировке последовательности из трёх элементов с помощью алгоритма сортировки вставками из раздела 1.1.

Пусть теперь мы сортируем n элементов $a_1, \ldots, a_n$. Каждая внутренняя вершина разрешающего дерева соответствует операции сравнения и снабжена пометкой вида $a_i : a_j$, указывающей, какие элементы надо сравнить ($1 \leqslant i, j \leqslant n$). Каждый лист разрешающего дерева снабжен пометкой $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$, где π — перестановка n элементов (см. разд. 6.1 по поводу перестановок). Для получения нижних оценок мы можем ограничиться случаем различных элементов, тогда результатом сортировки будет перестановка элементов в порядке возрастания.

Опишем, какой алгоритм сортировки соответствует данному разрешающему дереву. Надо пройти по дереву от корня до листа. Выбор направления (налево или направо) происходит так. Пусть в вершине написано $a_i : a_j$. Тогда надо идти налево, если $a_i \leqslant a_j$, и направо в противном случае. Если в листе, в который мы в итоге приходим, записана перестановка π, то результатом сортировки считаем последовательность $a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}$, которая должна быть неубывающей, если алгоритм правилен.

Каждая из $n!$ перестановок должна появиться хотя бы на одном листе разрешающего дерева (поскольку правильный алгоритм должен предусматривать все возможные порядки).

Нижняя оценка для худшего случая

Число сравнений в худшем случае для такого алгоритма равно высоте разрешающего дерева — максимальной длине пути в этом дереве от корня до листа. Следующая теорема дает нижнюю оценку на эту высоту.

Теорема 9.1. *Высота любого разрешающего дерева, сортирующего* n *элементов, есть* $\Omega(n \lg n)$.

Доказательство. Поскольку среди листьев разрешающего дерева должны быть представлены все перестановки n элементов, число этих листьев не менее $n!$. Поскольку двоичное дерево высоты h имеет не более 2^h листьев, имеем $n! \leqslant 2^h$. Логарифмируя это неравенство по основанию 2 и пользуясь неравенством $n! > (n/e)^n$, вытекающим из формулы Стирлинга (2.11), получаем, что

$$h \geqslant n \lg n - n \lg e = \Omega(n \lg n),$$

что и утверждалось. □

Следствие 9.2. Алгоритмы сортировки слиянием и с помощью кучи асимптотически оптимальны.

Доказательство. Они работают за время $O(n \lg n)$; в силу доказанной теоремы, эта оценка асимптотически неулучшаема. □

Упражнения

9.1-1 Какова наименьшая возможная глубина листа в разрешающем дереве алгоритма сортировки?

9.1-2 Докажите асимптотически точную оценку для $\lg(n!) = \sum_{k=1}^{n} \lg k$ без формулы Стирлинга, используя методы разд. 3.2.

9.1-3 Покажите, что не существует алгоритма сортировки, основанного на сравнениях, который работал бы за линейное время для половины из $n!$ возможных входных последовательностей длины n. Как изменится ответ, если в этой задаче заменить $1/2$ на $1/n$? На $1/2^n$?

9.1-4 Профессор разработал компьютер, который поддерживает "*тройные ветвления*": после одного-единственного сравнения $a_i : a_j$ управление может быть передано в одно из трёх мест программы, в зависимости от того, какое из соотношений выполнено: $a_i < a_j$, $a_i = a_j$ или $a_i > a_j$. Он надеется, что благодаря таким сравнениям сортировку n элементов можно провести асимптотически быстрее, чем за время $\Omega(n \lg n)$. Покажите, что профессор заблуждается.

9.1-5 Покажите, что для слияния двух отсортированных последовательностей из n элементов достаточно $2n-1$ сравнений в худшем случае.

9.1-6 Последовательность из n элементов, которую необходимо отсортировать, разбита на участки длины k. При этом любой элемент первого участка меньше любого элемента второго и т.д. (так что остаётся лишь отсортировать элементы внутри участков). Покажите, что такая сортировка потребует не менее $\Omega(n \lg k)$ сравнений в худшем случае. (Указание: недостаточно сослаться на необходимость n/k раз сортировать участок длиной k.)

9.2 Сортировка подсчётом

Алгоритм **сортировки подсчётом** (counting sort) применим, если каждый из n элементов сортируемой последовательности — целое положительное число в известном диапазоне (не превосходящее заранее известного k). Если $k = O(n)$, то алгоритм сортировки подсчётом работает за время $O(n)$.

Идея этого алгоритма в том, чтобы для каждого элемента x предварительно подсчитать, сколько элементов входной последовательности меньше x, после чего записать x напрямую в выходной массив в соответствии с этим числом (если, скажем, 17 элементов входного массива меньше x, то в выходном массиве x должен быть записан на место номер 18). Если в сортируемой последовательности могут присутствовать равные числа, эту схему надо слегка модифицировать, чтобы не записать несколько чисел на одно место.

В приводимом ниже псевдокоде используется вспомогательный массив $C[1\,..\,k]$ из k элементов. Входная последовательность записана в массиве $A[1\,..\,n]$, отсортированная последовательность записывается в массив $B[1\,..\,n]$.

```
Counting-Sort($A, B, k$)
1   for $i \leftarrow 1$ to $k$
2       do $C[i] \leftarrow 0$
3   for $j \leftarrow 1$ to $length[A]$
4       do $C[A[j]] \leftarrow C[A[j]] + 1$
5   ▷ $C[i]$ равно количеству элементов, равных $i$.
6   for $i \leftarrow 2$ to $k$
7       do $C[i] \leftarrow C[i] + C[i-1]$
8   ▷ $C[i]$ равно количеству элементов, не превосходящих $i$
9   for $j \leftarrow length[A]$ downto 1
10      do $B[C[A[j]]] \leftarrow A[j]$
11         $C[A[j]] \leftarrow C[A[j]] - 1$
```

Работа алгоритма сортировки подсчётом проиллюстрирована на рис. 9.2. После инициализации (строки 1–2) мы сначала помещаем в $C[i]$ количество элементов массива A, равных i (строки 3–4), а затем, находя частичные суммы последовательности $C[1], C[2], \ldots, C[k]$, — количество элементов, не превосходящих i (строки 6–7). Наконец, в строках 9–11 каждый из элементов

Рисунок 9.2 Работа алгоритма Counting-Sort, применённого к массиву $A[1\,..\,8]$, состоящему из натуральных чисел, не превосходящих $k = 6$. (а) Массив A и вспомогательный массив C после выполнения цикла в строках 3–4. (б) Массив C после выполнения цикла в строках 6–7. (в–д) Выходной массив B и вспомогательный массив C после одного, двух и трёх повторений цикла в строках 9–11. Зачернённые клетки соответствуют элементам массива, значения которым ещё не присвоены. (е) Массив B после окончания работы алгоритма.

массива A помещается на нужное место в массиве B. В самом деле, если все n элементов различны, то в отсортированном массиве число $A[j]$ должно стоять на месте номер $C[A[j]]$, ибо именно столько элементов массива A не превосходят $A[j]$; если в массиве A встречаются повторения, то после каждой записи числа $A[j]$ в массив B число $C[A[j]]$ уменьшается на единицу (строка 11), так что при следующей встрече с числом, равным $A[j]$, оно будет записано на одну позицию левее.

Оценим время работы алгоритма сортировки подсчётом. Циклы в строках 1–2 и 6–7 работают за время $O(k)$, циклы в строках 3–4 и 10–11 — за время $O(n)$, а весь алгоритм, стало быть, работает за время $O(k + n)$. Если $k = O(n)$, то время работы есть $O(n)$.

Для алгоритма сортировки подсчётом нижняя оценка разд. 9.1 — не препятствие, поскольку он не сравнивает сортируемые числа между собой, а использует их в качестве индексов массива.

Алгоритм сортировки подсчётом обладает важным свойством, называемым **устойчивостью** (it is stable). Именно, если во входном массиве присутствует несколько равных чисел, то в выходном массиве они стоят в том же порядке, что и во входном. Это свойство не имеет смысла, если в массиве записаны только числа сами по себе, но если вместе с числами записаны дополнительные данные, это оказывается важным. Более точно, представим себе, что мы сортируем не просто числа, а пары $\langle t, x\rangle$, где t — число от 1 до k, а x — произвольный объект, и хотим переставить их так, чтобы первые компоненты пар шли в неубывающем порядке. Описанный нами алгоритм позволяет это сделать, причём относительное расположение пар с равными первыми компонентами не меняется. Это свойство и называется устойчивостью. Мы увидим в следующем разделе, как оно применяется.

Упражнения

9.2-1 Следуя образцу рис. 9.2., покажите работу алгоритма Counting-Sort для случая $k = 7$ и $A = \langle 7, 1, 3, 1, 2, 4, 5, 7, 2, 4, 3 \rangle$.

9.2-2 Докажите, что алгоритм Counting-Sort является устойчивым.

9.2-3 Заменим строку 9 алгоритма Counting-Sort на такую:

```
9  for $j \leftarrow 1$ to $length[A]$
```

Покажите, что алгоритм остаётся правильным. Будет ли он устойчив?

9.2-4 Пусть на выходе алгоритма сортировки надо напечатать элементы входной последовательности в отсортированном порядке. Модифицируйте алгоритм Counting-Sort таким образом, чтобы он делал это, не используя массива B или иных массивов (помимо A и C). (Указание: свяжите в списки элементы массива A с одинаковым значением; где взять место для хранения указателей?).

9.2-5 Дано n целых чисел от 1 до k. Разработайте алгоритм, который подвергает эти данные предварительной обработке, а затем за время $O(1)$ отвечает на любой вопрос типа *"сколько чисел из данного набора лежит между a и b?"*. Время на предварительную обработку должно быть $O(n + k)$.

9.3 Цифровая сортировка

Алгоритм **цифровой сортировки** (radix sort) использовался в машинах для сортировки перфокарт (сейчас такие машины можно найти разве что в музеях). В картонных перфокартах специальный перфоратор пробивал дырки. В каждой из 80 колонок были места для 12 прямоугольных дырок. Вообще-то в одной колонке можно было пробить несколько дырок, их комбинация соответствовала символу (так что на карте было место для 80 символов), но цифры 0–9 кодировались одиночными дырками в строках 0–9 соответствующей колонки.

Сортировочной машине указывали столбец, по которому нужно произвести сортировку, и она раскладывала колоду перфокарт на

$$\begin{array}{ccccccc}
329 & & 720 & & 720 & & 329 \\
457 & & 355 & & 329 & & 355 \\
657 & & 436 & & 436 & & 436 \\
839 & & 457 & & 839 & & 457 \\
436 & \Rightarrow & 657 & \Rightarrow & 355 & \Rightarrow & 657 \\
720 & & 329 & & 457 & & 720 \\
355 & & 839 & & 657 & & 839 \\
 & & \uparrow & & \uparrow & & \uparrow
\end{array}$$

Рисунок 9.3 Цифровая сортировка последовательности из семи трёхзначных чисел. На вход подаются числа первого столбца. Далее показан порядок чисел после сортировки по третьей, второй и первой цифрам (вертикальные стрелки указывают, по какой цифре производилась сортировка).

10 стопок в зависимости от того, какая из дырок 0–9 была пробита в указанном столбце.

Как отсортировать колоду перфокарт с многозначными числами (разряд единиц в одном столбце, десятков — в предыдущем и т. д.)? Первое, что приходит в голову, — начать сортировку со старшего разряда. При этом получится 10 стопок, каждую из которых на следующем шаге придётся разбивать на 10 стопок, и так далее — получится много стопок перфокарт, в которых легко запутаться (см. упражнение 9.3-5).

Как ни странно, оказывается удобнее начать *с младшего разряда*, разложив колоду на 10 стопок в зависимости от того, где пробито отверстие в *"младшем"* столбце. Полученные 10 стопок надо после этого сложить в одну в таком порядке: сначала карты с 0, затем карты с 1, и т.д. Получившуюся колоду вновь рассортируем на 10 стопок, но уже в соответствии с разрядом десятков, сложим полученные стопки в одну колоду, и т. д.; если на перфокартах были записаны d-значные числа, то понадобится d раз воспользоваться сортировочной машиной. На рис. 9.3 изображено, как действует этот алгоритм, примененный к семи трёхзначным числам.

Важно, чтобы алгоритм, с помощью которого происходит сортировка по данному разряду, был устойчивым: карточки, у которых в данной колонке стоит одна и та же цифра, должны выйти из сортировочной машины в той же последовательности, в которой они туда подавались (само собой, при складывании 10 стопок в одну менять порядок карт в стопках тоже не следует).

В компьютерах цифровая сортировка иногда используется для упорядочения данных, содержащих несколько полей. Пусть, например, нам надо отсортировать последовательность дат. Это можно сделать с помощью любого алгоритма сортировки, сравнивая даты следующим образом: сравнить годы, если годы совпадают — сравнить месяцы, если совпадают и месяцы — сравнить числа. Вместо этого, однако, можно просто трижды отсортировать массив дат с помощью устойчивого алгоритма: сначала по дням, потом по месяцам, потом по годам.

Программу для цифровой сортировки написать легко. Мы предполагаем, что каждый элемент n-элементного массива A состоит из d цифр, причем цифра номер 1 — младший разряд, а цифра номер d — старший.

```
Radix-Sort($A, d$)
1  for $i \leftarrow 1$ to $d$
2       do отсортировать массив $A$ устойчивым
           алгоритмом по значению цифры номер $i$
```

Правильность алгоритма цифровой сортировки доказывается индукцией по номеру разряда (см. упр. 9.3-3). Время работы зависит от времени работы выбранного устойчивого алгоритма. Если цифры могут принимать значения от 1 до k, где k не слишком велико, то очевидный выбор — сортировка подсчётом. Для n чисел с d знаками от 0 до $k-1$ каждый проход занимает время $\Theta(n+k)$; поскольку мы делаем d проходов, время работы цифровой сортировки равно $\Theta(dn+kd)$. Если d постоянно и $k=O(n)$, то цифровая сортировка работает за линейное время.

При цифровой сортировке важно правильно выбрать основание системы счисления, поскольку от него зависит размер требуемой дополнительной памяти и время работы. Конкретный пример: пусть надо отсортировать миллион 64-битных чисел. Если рассматривать их как четырёхзначные числа в системе счисления с основанием 2^{16}, то при цифровой сортировке мы справимся с ними за четыре прохода, используя в процедуре Counting-Sort массив B размером 2^{16} (что немного по сравнению с размером сортируемого массива) Это выгодно отличается от сортировки сравнением, когда на каждое число приходится по $\lg n \approx 20$ операций. К сожалению, цифровая сортировка, опирающаяся на сортировку подсчётом, требует ещё одного массива (того же размера, что и сортируемый) для хранения промежуточных результатов, в то время как многие алгоритмы сортировки сравнением обходятся без этого. Поэтому, если надо экономить память, алгоритм быстрой сортировки может оказаться предпочтительнее.

Упражнения

9.3-1 Следуя образцу рис. 9.3, покажите, как происходит цифровая сортировка (по алфавиту) английских слов COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

9.3-2 Какие из следующих алгоритмов сортировки являются устойчивыми: сортировка вставками, сортировка слиянием, сортировка с помощью кучи, быстрая сортировка? Объясните, каким способом можно любой алгоритм сортировки превратить в устойчивый. Сколько при этом потребуется дополнительного времени и памяти?

9.3-3 Докажите по индукции, что алгоритм цифровой сортировки правилен. Где в вашем доказательстве используется устойчивость алгоритма сортировки цифр?

9.3-4 Объясните, как рассортировать n целых положительных чисел, не превосходящих n^2, за время $O(n)$.

9.3-5* Пусть мы сортируем перфокарты с помощью сортировочной машины, начиная со старшего разряда. Сколько раз придётся запустить машину (в худшем случае) для сортировки d-значных чисел? Какое максимальное количество стопок карт придётся одновременно хранить по ходу дела?

9.4 Сортировка вычёрпыванием

Алгоритм **сортировки вычёрпыванием** (bucket sort) работает за линейное (среднее) время. Как и сортировка подсчётом, сортировка вычёрпыванием годится не для любых исходных данных: говоря о линейном среднем времени, мы предполагаем, что на вход подаётся последовательность независимых случайных чисел, равномерно распределённых на промежутке $[0;1)$ (определение равномерного распределения дано в разд. 6.2).

[Заметим, что этот алгоритм — детерминированный (не использует генератора случайных чисел); понятие случайности возникает лишь при анализе времени его работы.]

Идея алгоритма состоит в том, что промежуток $[0;1)$ делится на n равных частей, после чего для чисел из каждой части выделяется свой ящик-**черпак** (bucket), и n подлежащих сортировке чисел раскладываются по этим ящикам. Поскольку числа равномерно распределены на отрезке $[0;1)$, следует ожидать, что в каждом ящике их будет немного. Теперь отсортируем числа в каждом ящике по отдельности и пройдёмся по ящикам в порядке возрастания, выписывая попавшие в каждый из них числа также в порядке возрастания.

Будем считать, что на вход подается n-элементный массив A, причем $0 \leqslant A[i] < 1$ для всех i. Используется также вспомогательный массив $B[0\,..\,n-1]$, состоящий из списков, соответствующих ящикам. Алгоритм использует операции со списками, которые опи-

Рисунок 9.4 Работа алгоритма Bucket-Sort. (а) На вход подан массив $A[1\,..\,10]$. (б) Массив списков $B[0\,..\,9]$ после выполнения строки 5. Список с индексом i содержит числа, у которых первый знак после запятой есть i. Отсортированный массив получится, если последовательно выписать списки $B[0], \ldots, B[9]$.

саны в разд. 11.2.

```
Bucket-Sort($A$)
1  $n \leftarrow length[A]$
2  for $i \leftarrow 1$ to $n$
3        do добавить $A[i]$ к списку $B[\lfloor nA[i] \rfloor]$
4  for $i \leftarrow 0$ to $n-1$
5        do отсортировать список $B[i]$ (сортировка вставками)
6  соединить списки $B[0], B[1], \ldots, B[n-1]$ (в указанном порядке)
```

На рис. 9.4 показана работа этого алгоритма на примере массива из 10 чисел.

Чтобы показать, что алгоритм сортировки вычёрпыванием правилен, рассмотрим два числа $A[i]$ и $A[j]$. Если они попали в разные ящики, то меньшее из них попало в ящик с меньшим номером, и в выходной последовательности оно окажется раньше; если они попали в один ящик, то после сортировки содержимого ящика меньшее число будет также предшествовать большему.

Проанализируем время работы алгоритма. Операции во всех строках, кроме пятой, требуют (общего) времени $O(n)$. Просмотр всех ящиков также занимает время $O(n)$. Таким образом, нам остаётся только оценить время сортировки вставками внутри ящиков.

Пусть в ящик $B[i]$ попало n_i чисел (n_i — случайная величина). Поскольку сортировка вставками работает за квадратичное время, математическое ожидание длительности сортировки чисел в ящике номер i есть $O(\mathsf{M}[n_i^2])$, а математическое ожидание суммарного времени сортировки во всех ящиках есть

$$\sum_{i=0}^{n-1} O(\mathsf{M}[n_i^2]) = O\left(\sum_{i=0}^{n-1} \mathsf{M}[n_i^2]\right). \tag{9.1}$$

Найдём функцию распределения случайных величин n_i. Поскольку числа распределены равномерно, а величины всех отрезков равны, вероятность того, что данное число попадет в ящик номер i, равна $1/n$. Стало быть, мы находимся в ситуации примера из разд. 6.6.2 с шарами и урнами: у нас n шаров-чисел, n урн-ящиков, и вероятность попадания данного шара в данную урну равна $p = 1/n$. Поэтому числа n_i распределены биномально: вероятность того, что $n_i = k$, равна $C_n^k p^k (1-p)^{n-k}$, математическое ожидание равно $\mathsf{M}[n_i] = np = 1$, и дисперсия равна $\mathsf{D}[n_i] = np(1-p) = 1 - 1/n$. Из формулы (6.30) имеем:

$$\mathsf{M}[n_i^2] = \mathsf{D}[n_i] + \mathsf{M}^2[n_i] = 2 - \frac{1}{n} = \Theta(1).$$

Подставляя эту оценку в (9.1), получаем, что математическое ожидание суммарного времени сортировки всех ящиков есть $O(n)$, так что математическое ожидание времени работы алгоритма сортировки вычёрпыванием в самом деле линейно зависит от количества чисел.

Упражнения

9.4-1 Следуя образцу рис. 9.4, покажите, как работает алгоритм BUCKET-SORT для массива $A = \langle 0.79, 0.13, 0.16, 0.64, 0.39, 0.20, 0.89, 0.53, 0.71, 0.42\rangle$.

9.4-2 Каково время работы алгоритма сортировки вычёрпыванием в худшем случае? Придумайте его простую модификацию, сохраняющую линейное среднее время работы и снижающую время работы в худшем случае до $O(n \lg n)$.

9.4-3* Дано n независимых случайных точек с координатами $(x_i; y_i)$, равномерно распределённых в круге радиуса 1 с центром в начале координат (это означает, что вероятность найти точку в какой-то области пропорциональна площади этой области). Разработайте алгоритм, располагающий точки в порядке возрастания расстояния от центра и имеющий среднее время работы $\Theta(n)$. (Указание: воспользуйтесь сортировкой вычёрпыванием, но позаботьтесь о том, чтобы площади ящиков были равны).

9.4-4* Пусть X — случайная величина. Ее **функция распределения** (probability distribution function) определяется формулой $P(x) =$

$\mathsf{P}\{X \leqslant x\}$. Предположим, что на вход алгоритма поступает последовательность из n чисел, которые являются независимыми случайными величинами с функцией распределения P. Функция P непрерывна и может быть вычислена за время $O(1)$. Как отсортировать такую последовательность, чтобы среднее время сортировки линейно зависело от n?

Задачи

9-1 Нижние оценки для среднего числа сравнений

В этой задаче мы докажем, что среднее время работы любого детерминированного или вероятностного алгоритма сортировки n чисел, основанного на сравнениях, есть $\Omega(n \lg n)$. Начнем с того, что рассмотрим детерминированный алгоритм A, основанный на сравнениях; пусть T_A — его разрешающее дерево. Мы предполагаем, что все перестановки входной последовательности равновероятны.

а. Напишем на каждом листе дерева T_A вероятность того, что алгоритм завершится в этом листе. Покажите, что в $n!$ листах написано $1/n!$, а в остальных листах написан нуль.

б. Обозначим через $D(T)$ сумму глубин всех листьев в двоичном дереве T (мы предполагаем, что каждая вершина либо является листом, либо имеет двух детей — так устроены разрешающие деревья). Пусть T — дерево с $k > 1$ листьями, и пусть через LT и RT обозначены левое и правое поддеревья. Покажите, что $D(T) = D(LT) + D(RT) + k$.

в. Пусть $d(m)$ — наименьшее значение числа $D(T)$ среди всех деревьев T с m листьями. Покажите, что $d(k) = \min_{1 \leqslant i \leqslant k-1}\{d(i) + d(k-i) + k\}$. (Указание: дерево LT может содержать от 1 до $k-1$ листьев.)

г. Покажите, что для фиксированного k выражение $i \lg i + (k-i)\lg(k-i)$ достигает минимума на отрезке от 1 до $k-1$ в точке $i = k/2$. Выведите отсюда, что $d(k) = \Omega(k \lg k)$.

д. Покажите, что $D(T_A) = \Omega(n! \lg(n!))$, и выведите отсюда, что время сортировки n чисел с помощью разрешающего дерева T_A, усреднённое по всем перестановкам на входе, есть $\Omega(n \lg n)$.

Теперь рассмотрим вероятностный алгоритм B, основанный на сравнениях. Его можно описать с помощью разрешающего дерева, в котором бывают узлы двух типов: соответствующие сравнениям и случайным выборам. В узлах второго типа происходит вызов процедуры RANDOM$(1, r)$; у такого узла r детей, каждый из которых выбирается с равной вероятностью. (Для разных узлов число r может быть разным.)

е. Пусть имеется вероятностный алгоритм сортировки B. Для каждой перестановки на входе найдём математическое ожидание числа сравнений. Усредним эти ожидания по всем входам; получится некоторое число T. Покажите, что существует детерминированный алгоритм A, у которого среднее (по всем перестановкам на входе) число сравнений не превосходит T. Выведите отсюда, что для любого вероятностного алгоритма максимальное (по всем входам) математическое ожидание числа сравнений есть $\Omega(n \lg n)$.

(Указание. Если среднее по всем входам и всем вариантам случайных чисел равно T, то существует такой набор случайных чисел, при котором среднее по всем входам не больше T.)

9-2 Сортировка без дополнительной памяти за линейное время

а. Пусть нам дан массив записей, который необходимо отсортировать по ключу, принимающему значение 0 или 1. Придумайте простой алгоритм, осуществляющий такую сортировку за линейное время и использующий дополнительную память $O(1)$ (иными словами, объем дополнительной памяти не зависит от размеров сортируемого массива).

б. Можно ли воспользоваться алгоритмом из пункта (а) для цифровой сортировки по b-битному ключу за время $O(bn)$? Объясните свой ответ.

в. Пусть n записей надо отсортировать по ключу, принимающему целые значения от 1 до k. Как модифицировать сортировку подсчётом, чтобы можно было отсортировать эти записи за время $O(n + k)$, и при этом объем используемой памяти (помимо сортируемого массива) был $O(k)$? (Указание. Начните со случая $k = 3$.)

Замечания

Анализ алгоритмов сортировки с помощью разрешающих деревьев был предложен Фордом и Джонсоном [72]. В фундаментальной книге Кнута [123], посвященной сортировке, рассматриваются многочисленные варианты этой задачи и доказывается нижняя оценка числа сравнений (приведённая в этой главе). Нижние оценки для задачи сортировки и различных обобщений разрешающих деревьев подробно изучались Бен-Ором [23].

Согласно Кнуту, заслуга изобретения сортировки подсчётом (1954 год) и её комбинации с цифровой сортировкой принадлежит Сьюарду (H. H. Seward). Сама же цифровая сортировка, видимо, давно применялась для сортировки перфокарт. Кнут утверждает, что первое печатное описание этого метода появилось в 1929 году в качестве составной части руководства по оборудованию для работы с перфокартами, написанного Комри (L. J. Comrie). Сортировка вычёрпыванием используется с 1956 года, когда Айзек (E. J. Isaac) и Синглетон (R. C. Singleton) предложили основную идею.

10 Медианы и порядковые статистики

В этой главе мы рассматриваем такую задачу: дано множество из n чисел; найти тот его элемент, который будет i-м по счёту, если расположить элементы множества в порядке возрастания. В англоязычной литературе такой элемент называется i-й **порядковой статистикой** (order statistic). Например, **минимум** (minimum) — это порядковая статистика номер 1, а **максимум** (maximum) — порядковая статистика номер n. **Медианой** (median) называется элемент множества, находящийся (по счёту) посередине между минимумом и максимумом. Точнее говоря, если n нечётно, то медиана — это порядковая статистика номер $i = (n+1)/2$, а если n чётно, то медиан даже две: с номерами $i = n/2$ и $i = n/2+1$. Можно ещё сказать, что, независимо от чётности n, медианы имеют номер $i = \lfloor (n+1)/2 \rfloor$ и $i = \lceil (n+1)/2 \rceil$. В дальнейшем мы будем называть медианой меньшую из двух (если их две).

Для удобства мы будем считать, что множество, в котором мы ищем порядковые статистики, состоит из различных элементов, хотя практически всё, что мы делаем, переносится на ситуацию, когда во множестве есть повторяющиеся элементы. **Задача выбора элемента с данным номером** (selection problem) состоит в следующем:

Дано: Множество A из n различных элементов и целое число i, $1 \leqslant i \leqslant n$.

Найти: Элемент $x \in A$, для которого ровно $i-1$ элементов множества A меньше x.

Эту задачу можно решить за время $O(n \lg n)$: отсортировать числа, после чего взять i-й элемент в полученном массиве. Есть, однако, и более быстрые алгоритмы.

В разделе 10.1 мы рассмотрим простейший случай: нахождение максимального и минимального элементов. Общая задача более интересна; ей посвящены два следующих раздела. В разделе 10.2 мы рассматриваем удобный на практике вероятностный алгоритм, который ищет порядковую статистику за время $O(n)$ в среднем (имеется в виду математическое ожидание времени его работы на любом входе). В разделе 10.3 мы рассматриваем (представляющий скорее теоретический интерес) детерминированный алгоритм, требующий времени $O(n)$ в худшем случае.

10.1 Минимум и максимум

Сколько сравнений необходимо, чтобы во множестве из n чисел найти наименьшее? За $n - 1$ сравнений это сделать легко: надо последовательно перебирать все числа, храня значение наименьшего числа из уже просмотренных. Запишем этот алгоритм, считая, что числа заданы в виде массива A длины n.

```
MINIMUM(A)
1  min ← A[1]
2  for i ← 2 to length[A]
3       do if min > A[i]
4             then min ← A[i]
5  return min
```

Разумеется, аналогичным образом можно найти и максимум.

Можно ли найти минимум еще быстрее? Нет, и вот почему. Рассмотрим алгоритм нахождения наименьшего числа как турнир среди n чисел, а каждое сравнение — как матч, в котором меньшее число побеждает. Чтобы победитель был найден, каждое из остальных чисел должно проиграть по крайней мере один матч, так что меньше $n-1$ сравнений быть не может, и алгоритм MINIMUM оптимален по числу сравнений.

Интересный вопрос, связанный с этим алгоритмом — нахождение математического ожидания числа исполнений строки 4. В задаче 6-2 требуется показать, что эта величина есть $\Theta(\lg n)$.

Одновременный поиск минимума и максимума

Иногда бывает нужно найти одновременно минимальный и максимальный элементы множества. Представим себе программу, которая должна уменьшить рисунок (набор точек, заданных своими координатами) так, чтобы он уместился на экране. Для этого нужно найти максимум и минимум по каждой координате.

Если мы попросту найдем сначала минимум, а потом максимум, затратив на каждый из них по $n - 1$ сравнений, то всего будет $2n - 2$ сравнения, что асимптотически оптимально. Можно, однако, решить эту задачу всего за $3\lceil n/2 \rceil - 2$ сравнений. Именно, будем хранить значения максимума и минимума уже просмотренных чисел, а очередные числа будем обрабатывать по два таким образом: сначала сравним два очередных числа друг с другом, а затем большее из них сравним с максимумом, а меньшее — с минимумом.

При этом на обработку двух элементов мы затратим три сравнения вместо четырёх (кроме первой пары, где понадобится всего одно сравнение).

Упражнения

10.1-1 Покажите, что второе по величине число из n данных можно найти в худшем случае за $n + \lceil \lg n \rceil - 2$ сравнения. (Указание. Ищите это число вместе с наименьшим.)

10.1-2* Покажите, что для одновременного нахождения наибольшего и наименьшего из n чисел в худшем случае необходимо не менее $\lceil 3n/2 \rceil - 2$ сравнений. (Указание. В каждый момент элементы делятся на четыре группы: (1) непросмотренные (которые могут оказаться и минимальными, и максимальными); (2) те, что ещё могут оказаться минимальными, но заведомо не максимальны; (3) те, что ещё могут оказаться максимальными, но не минимальными; (4) отброшенные (которые заведомо не минимальны и не максимальны). Пусть a_1, a_2, a_3, a_4 — количество элементов каждой группы. Как меняются эти числа при сравнениях?)

10.2 Выбор за линейное в среднем время

Хотя общая задача выбора выглядит более сложной, чем задача о минимуме или максимуме, её, как ни странно, тоже можно решить за время $\Theta(n)$. В этом разделе мы рассмотрим вероятностный алгоритм RANDOMIZED-SELECT для решения этой задачи, действующий по схеме *"разделяй и властвуй"*. Он аналогичен алгоритму быстрой сортировки из главы 8: массив разбивается на меньшие части. Однако алгоритм быстрой сортировки, разбив массив на два куска, обрабатывает оба, а алгоритм RANDOMIZED-SELECT — только один из этих кусков. Поэтому он быстрее: среднее время быстрой сортировки есть $\Theta(n \lg n)$, в то время как алгоритм RANDOMIZED-SELECT работает в среднем за время $\Theta(n)$.

Алгоритм RANDOMIZED-SELECT использует процедуру RANDOMIZED-PARTITION, описанную в разделе 8.3, поведение которой (и, стало быть, всего алгоритма) зависит от датчика случайных чисел. Вызов RANDOMIZED-SELECT(A, p, r, i) возвращает i-й по счёту в порядке возрастания элемент массива $A[p \,..\, r]$.

```
Randomized-Select($A, p, r, i$)
1  if $p = r$
2      then return $A[p]$
3  $q \leftarrow$ Randomized-Partition($A, p, r$)
4  $k \leftarrow q - p + 1$
5  if $i \leqslant k$
6      then return Randomized-Select($A, p, q, i$)
7      else  return Randomized-Select($A, q + 1, r, i - k$)
```

После исполнения процедуры Randomized-Partition в строке 3 массив $A[p\,..\,r]$ состоит из двух непустых частей $A[p\,..\,q]$ и $A[q + 1\,..\,r]$, причём всякий элемент $A[p\,..\,q]$ меньше всякого элемента $A[q{+}1\,..\,r]$. В строке 4 вычисляется количество элементов в массиве $A[p\,..\,q]$. Дальнейшее зависит от того, в каком из этих двух массивов лежит i-й по величине элемент массива $A[p\,..\,r]$. Если $i \leqslant k$, то нужный нам элемент лежит в левой части (массиве $A[p\,..\,q]$), откуда и извлекается в результате (рекурсивного) вызова Randomized-Select в строке 6. Если же $i > k$, то все k элементов левой части ($A[p\,..\,q]$) заведомо меньше искомого, который можно найти как $(i - k)$-й по счёту элемент массива $A[q + 1\,..\,r]$ (строка 7).

Время работы алгоритма Randomized-Select (с учётом времени работы процедуры Randomized-Partition) в худшем случае есть $\Theta(n^2)$, даже если мы ищем всего лишь минимум: в самом деле, при особом невезении может оказаться, что мы все время разбиваем массив возле наибольшего оставшегося элемента. Однако случайный выбор гарантирует, что для любого входа среднее время работы алгоритма будет невелико.

Докажем это. Рассмотрим для каждой из возможных перестановок n элементов математическое ожидание времени работы алгоритма (среднее по всем случайным выборам). Пусть $T(n)$ — максимальное из этих ожиданий. Как мы отмечали в разделе 8.4, после вызова Randomized-Partition левая часть разбиения содержит 1 элемент с вероятностью $2/n$, и содержит $i > 1$ элементов с вероятностью $1/n$ для любого i от 2 до $n - 1$. Можно считать, что $T(n)$ монотонно возрастает с ростом n (если это не так, заменим $T(n)$ на $T'(n) = \max_{1 \leqslant i \leqslant n} T(i)$ и повторим все рассуждения для T'). Заметим, что худшим случаем будет тот, когда i-й по счёту элемент попадает в бóльшую из двух половин, на которые разбивается массив. Отсюда имеем:

$$\begin{aligned}
T(n) &\leqslant \frac{1}{n}\left(T(\max(1, n-1)) + \sum_{k=1}^{n-1} T(\max(k, n-k))\right) + O(n) \\
&\leqslant \frac{1}{n}\left(T(n-1) + 2\sum_{k=\lceil n/2\rceil}^{n-1} T(k)\right) + O(n) \\
&= \frac{2}{n}\sum_{k=\lceil n/2\rceil}^{n-1} T(k) + O(n)
\end{aligned}$$

(мы сгруппировали одинаковые слагаемые при переходе от первой строки ко второй и отбросили $T(n-1)/n$ при переходе от второй строки к третьей, так как в худшем случае $T(n-1) = O(n^2)$, и тем самым слагаемое $T(n-1)/n$ можно включить в $O(n)$).

Теперь докажем, что $T(n) \leqslant cn$, рассуждая по индукции (как выбрать c, мы скажем после). В самом деле, пусть неравенство $T(j) \leqslant cj$ верно для всех $j < n$. Тогда, используя формулу для суммы арифметической прогрессии, получаем, что

$$\begin{aligned}
T(n) &\leqslant \frac{2}{n}\sum_{k=\lceil n/2\rceil}^{n-1} ck + O(n) \\
&\leqslant \frac{2c}{n}\cdot\frac{n}{2}\cdot\frac{n/2+1+n-1}{2} + O(n) \\
&= \frac{3cn}{4} + O(n) \leqslant cn,
\end{aligned}$$

если выбрать c столь большим, чтобы $c/4$ превосходило константу, подразумеваемую в слагаемом $O(n)$.

Стало быть, любая порядковая статистика, и медиана в том числе, может быть найдена за линейное в среднем время.

Упражнения

10.2-1 Напишите версию процедуры Randomized-Select, не использующую рекурсии.

10.2-2 Предположим, что мы используем алгоритм Randomized-Select для выбора наименьшего элемента из массива $A = \langle 3, 2, 9, 0, 7, 5, 4, 8, 6, 1\rangle$. Опишите последовательность разбиений, соответствующую худшему случаю.

10.2-3 Будет ли алгоритм Randomized-Select правильно работать, если в массиве A есть равные элементы (как мы помним, в этом случае процедура Randomized-Partition разбивает $A[p\,..\,r]$

на части $A[p\,..\,q]$ и $A[q+1\,..\,r]$ таким образом, что всякий элемент $A[p\,..\,q]$ не превосходит всякого элемента $A[q+1\,..\,r]$)?

10.3 Выбор за линейное в худшем случае время

Теперь рассмотрим (детерминированный) алгоритм Select, решающий задачу о порядковых статистиках за время $O(n)$ в худшем случае. Как и Randomized-Select, этот алгоритм основан на последовательном разбиении массива на меньшие части; в данном случае, однако, мы гарантируем, что выбранное разбиение не является неудачным. Алгоритм Select использует (детерминированную) процедуру Partition (см. описание алгоритма быстрой сортировки в разд. 8.1), модифицированную таким образом, чтобы элемент, с которым сравнивают, задавался как параметр.

Алгоритм Select находит i-й по порядку элемент в массиве размера $n > 1$ следующим образом:

1. Разбить n элементов массива на $\lfloor n/5 \rfloor$ групп по 5 элементов и (возможно) одну группу, в которой менее пяти элементов.
2. Найти медиану каждой из $\lceil n/5 \rceil$ групп (для чего отсортировать группу вставками).
3. Вызвав (рекурсивно) процедуру Select, найти число x, являющееся медианой найденных $\lceil n/5 \rceil$ медиан.
4. Вызвав модифицированную процедуру Partition, разбить массив относительно найденной *"медианы медиан"*. Пусть k — количество элементов в нижней части этого разбиения (в верхней части, стало быть, $n-k$).
5. Вызвав (рекурсивно) процедуру Select, найти i-ую порядковую статистику нижней части разбиения, если $i \leqslant k$, или $(i-k)$-ю порядковую статистику верхней части разбиения, если $i > k$.

Оценим время работы алгоритма Select. Для начала выясним, сколько чисел заведомо будут больше *"медианы медиан"* x (см. рис. 10.1). Не менее половины медиан, найденных на втором шаге, будут больше или равны x. Стало быть, по крайней мере половина из $\lceil n/5 \rceil$ групп даст по три числа, бо́льших x, за двумя возможными исключениями: группа, содержащая x, и последняя неполная группа. Тем самым имеется не менее

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geqslant \frac{3n}{10} - 6.$$

элементов, заведомо больших x, и точно так же получаем, что имеется не менее $3n/10 - 6$ элементов, заведомо меньших x. Значит, алгоритм Select, рекурсивно вызываемый на пятом шаге, будет обрабатывать массив длиной не более $7n/10 + 6$.

Рисунок 10.1 Анализ алгоритма Select. Элементы массива (их n) изображены кружками, каждый столбец — группа из 5 или меньше элементов, белые кружки — медианы групп. *"Медиана медиан"* обозначена буквой x. Стрелки идут от бо́льших чисел к меньшим. Видно, что в каждом из полных столбцов правее x имеется три числа, бо́льших x, и что в каждом из столбцов левее x имеется три числа, меньших x. Множество чисел, заведомо бо́льших x, выделено серым.

Пусть теперь $T(n)$ — время работы алгоритма Select на массиве из n элементов в худшем случае. Первый, второй и четвертый шаги выполняются за время $O(n)$ (на втором шаге мы $O(n)$ раз сортируем массивы размером $O(1)$), третий шаг выполняется за время не более $T(\lceil n/5\rceil)$, а пятый шаг, по доказанному, — за время, не превосходящее $T(\lfloor 7n/10+6\rfloor)$ (как и раньше, можно предполагать, что $T(n)$ монотонно возрастает с ростом n). Стало быть,

$$T(n) \leqslant T(\lceil n/5\rceil) + T(\lfloor 7n/10+6\rfloor) + O(n).$$

Поскольку сумма коэффициентов при n в правой части ($1/5+7/10 = 9/10$) меньше единицы, из этого рекуррентного соотношения вытекает, что $T(n) \leqslant cn$ для некоторой константы c. Это можно доказать по индукции. В самом деле, предполагая, что $T(m) \leqslant cm$ для всех $m < n$, имеем

$$\begin{aligned} T(n) &\leqslant c(\lceil n/5\rceil) + c(\lfloor 7n/10+6\rfloor) + O(n) \\ &\leqslant c(n/5+1) + c(7n/10+6) + O(n) \\ &\leqslant 9cn/10 + 7c + O(n) = \\ &= cn - c(n/10-7) + O(n). \end{aligned}$$

При подходящем выборе c это выражение будет не больше cn при всех $n > 70$ (надо, чтобы $c(n/10-6)$ превосходило коэффициент, подразумеваемый в $O(n)$). Таким образом, индуктивный переход возможен при $n > 70$ (заметим ещё, что при таких n выражения $\lceil n/5\rceil$ и $\lfloor 7n/10+6\rfloor$ меньше n).

Увеличив c ещё (если надо), можно добиться того, чтобы $T(n)$ не превосходило cn и при всех $n \leqslant 70$, что завершает рассуждение по индукции. Стало быть, алгоритм Select работает за линейное время (в худшем случае).

Отметим, что алгоритмы Select и Randomized-Select, в отличие от описанных в главе 9 алгоритмов сортировки за линейное время, используют только попарные сравнения элементов массива и применимы для произвольного упорядоченного множества. Эти алгоритмы асимптотически эффективнее очевидного подхода *"упорядочи множество и выбери нужный элемент"*, поскольку всякий алгоритм сортировки, использующий только попарные сравнения, требует времени $\Omega(n \lg n)$ не только в худшем случае (раздел 9.1), но и в среднем (задача 9-1).

Упражнения

10.3-1 Будет ли алгоритм Select работать за линейное время, если разбивать массив на группы не из пяти, а из семи элементов? Покажите, что для групп из трёх элементов рассуждение не проходит.

10.3-2 Пусть x — *"медиана медиан"* в алгоритме Select (массив содержит n элементов). Покажите, что при $n \geqslant 38$ количество элементов, бо́льших x (так же как и количество элементов, меньших x) не меньше $\lceil n/4 \rceil$.

10.3-3 Модифицируйте алгоритм быстрой сортировки так, чтобы он работал за время $O(n \lg n)$ в худшем случае.

10.3-4* Пусть алгоритм выбора i-го по счёту элемента использует только попарные сравнения. Покажите, что с помощью тех же сравнений можно в качестве побочного результата получить списки элементов, меньших искомого, а также бо́льших искомого.

10.3-5 Пусть у нас есть какой-то алгоритм, находящий медиану за линейное в худшем случае время. Используя его в качестве подпрограммы, разработайте простой алгоритм, решающий задачу нахождения произвольной порядковой статистики за линейное время.

10.3-6 Под **k-квантилями** (k-th quantiles) множества из n чисел мы понимаем $k-1$ его элементов, обладающих следующим свойством: если расположить элементы множества в порядке возрастания, то квантили будут разбивать множество на k равных (точнее, отличающихся не более чем на один элемент) частей. Разработайте алгоритм, который за время $O(n \lg k)$ находит k-квантили данного множества.

10.3-7 Разработайте алгоритм, который по заданному k находит в данном множестве S его k элементов, менее всего отстоящих от медианы. Число операций должно быть $O(|S|)$.

Рисунок 10.2 Как провести с востока на запад магистраль, чтобы суммарная длина подводящих трубопроводов была минимальна?

10.3-8 Пусть $X[1\,..\,n]$ и $Y[1\,..\,n]$ — два возрастающих массива. Разработайте алгоритм, находящий за время $O(\lg n)$ медиану множества, полученного объединением элементов этих массивов.

10.3-9 Профессор консультирует нефтяную компанию, которой требуется провести магистральный нефтепровод в направлении строго с запада на восток через нефтеносное поле, на котором расположены n нефтяных скважин. От каждой скважины необходимо подвести к магистрали трубопровод по кратчайшему пути (строго на север или на юг, рис. 10.2). Координаты всех скважин профессору известны; необходимо выбрать местоположение магистрали, чтобы сумма длин всех трубопроводов, ведущих от скважин к магистрали, была минимальна. Покажите, что оптимальное место для магистрали можно найти за линейное время.

Задачи

10-1 Сортировка i наибольших элементов

Дано множество из n чисел; требуется выбрать из них i наибольших и отсортировать (пользуясь только попарными сравнениями). Для каждого из приведенных ниже подходов разработайте соответствующий алгоритм и выясните, как зависит от n и i время работы этих алгоритмов в худшем случае.

а Отсортировать все числа и выписать i наибольших.

б Поместить числа в очередь с приоритетами и вызвать i раз процедуру EXTRACT-MAX.

в Найти с помощью алгоритма раздела 10.3 i-е по величине число (считая от наибольшего), разбить массив относительно него и отсортировать i наибольших чисел.

10-2 Взвешенная медиана

Пусть дано n различных чисел $x_1, \ldots, x_n$, и пусть каждому x_i сопоставлено положительное число (*"вес"*) w_i, причём сумма всех весов равна 1. **Взвешенной медианой** (weighted median) называется такое число x_k, что

$$\sum_{x_i<x_k} w_i \leqslant \frac{1}{2} \quad \text{и} \quad \sum_{x_i>x_k} w_i \leqslant \frac{1}{2}.$$

а Покажите, что если все веса равны $1/n$, то взвешенная медиана совпадает с обычной.

б Как найти взвешенную медиану n чисел с помощью сортировки за время $O(n \lg n)$ в худшем случае?

в Как модифицировать алгоритм Select (раздел 10.3), чтобы он искал взвешенную медиану за время $\Theta(n)$ в худшем случае?

Задача о выборе места для почты (post-office location problem) состоит в следующем. Дано n точек $p_1, \ldots, p_n$ и n положительных весов $w_1, \ldots, w_n$; требуется найти точку p (не обязательно совпадающую с одной из p_i), для которой выражение $\sum_{i=1}^{n} w_i d(p, p_i)$ будет минимально (через $d(a, b)$ обозначается расстояние между точками a и b).

д Покажите, что в одномерном случае (точки — вещественные числа, $d(a, b) = |a - b|$) взвешенная медиана будет решением этой задачи.

е Найдите оптимальное решение в двумерном случае (точки — пары вещественных чисел), если расстояние между точками $a = (x_1, y_1)$ и $b = (x_2, y_2)$ задается *"в L_1-метрике"*: $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$ (американцы называют такую метрику Manhattan distance, по названию района Нью-Йорка, разбитого улицами на прямоугольные кварталы)

10-3 Нахождение i-го по величине элемента при малых i

Пусть $T(n)$ — время работы процедуры Select, примененной к массиву из n чисел; в худшем случае $T(n) = \Theta(n)$, но коэффициент при n, подразумеваемый в этом обозначении, довольно велик. Если i мало по сравнению с n, то отобрать i-ый по величине элемент можно быстрее.

а Опишите алгоритм, который находит i-й по величине элемент в множестве из n чисел, делая $U_i(n)$ сравнений, причём

$$U_i(n) = \begin{cases} T(n) & \text{если } n \leqslant 2i, \\ n/2 + U_i(\lfloor n/2 \rfloor) + T(2i+1) & \text{иначе.} \end{cases}$$

(Указание: сделайте $\lfloor n/2 \rfloor$ попарных сравнений и отберите i наименьших среди меньших элементов пар.)

б Покажите, что $U_i(n) = n + O(T(2i)\log(n/i))$.

в Покажите, что, при постоянном i, имеем $U_i(n) = n + O(\lg n)$. [Та же оценка получится, если построить кучу из главы 7, а затем i раз выбрать из неё минимальный элемент.]

г Пусть $i = n/k$, причём $k \geqslant 2$; покажите, что $U_i(n) = n + O(T(2n/k)\lg k)$.

Замечания

Алгоритм для нахождения медианы за линейное в худшем случае время предложили Блюм, Флойд, Пратт, Ривест и Тарьян [29]. Вероятностный алгоритм с линейным средним временем работы принадлежит Хоару [97]. Флойд и Ривест [70] разработали усовершенствованную версию этого алгоритма, в которой граница разбиения определяется по небольшой случайной выборке.

III Структуры данных

Введение

В программировании часто приходится иметь дело с множествами, меняющимися в процессе выполнения алгоритма. В следующих пяти главах мы рассмотрим структуры данных, предназначенные для хранения изменяющихся (**динамических**, по-английски dynamic) множеств.

Разные алгоритмы используют разные операции. Нередко, например, требуется лишь добавлять и удалять элементы в множество, а также проверять, принадлежит ли множеству данный элемент. Структура данных, поддерживающая такие операции, называется **словарём** (dictionary). В других ситуациях могут понадобиться более сложные операции. Например, очереди с приоритетами, о которых шла речь в главе 7 в связи с кучами, разрешают выбирать и удалять *наименьший* элемент (помимо добавления элементов). Понятно, что выбор реализации динамического множества зависит от того, какие операции с ним нам потребуются.

Элементы множеств

Обычно элемент динамического множества — это запись, содержащая различные поля. Часто одно из полей рассматривается как **ключ** (key), предназначенный для идентификации элемента, а остальные поля — как **дополнительная информация** (satellite data), хранящаяся вместе с ключом. Элемент множества ищется по ключу; когда элемент, мы можем прочесть или изменить дополнительную информацию, имеющуюся в этом элементе. Во многих случаях все ключи различны, и тогда множество можно рассматривать как функцию, которая с каждым (существующим) ключом сопоставляет некоторую дополнительную информацию.

Многие способы реализации множеств требуют, чтобы вместе с каждым ключом хранились не только дополнительные данные, но и некоторая служебная информация (например, указатели на другие элементы множества)

Часто на множестве ключей имеется естественный линейный порядок (например, ключи могут быть действительными числами или словами, на которых есть лексикографический порядок). В этом случае можно говорить, например, о наименьшем элементе множества или об элементе, непосредственно следующем за данным.

Мы предполагаем, что элементы множества хранятся в некоторой общей области памяти и для каждого имеется указатель, который позволяет получить доступ к этому элементу. Обычно в качестве указателя выступает просто адрес в памяти; если язык программирования этого не предусматривает, указателем может быть индекс в массиве.

Операции над множествами

Операции над множествами делятся на **запросы** (queries), которые не меняют множества, и **операции, меняющие множество** (modifying operations). Типичные операции с множествами таковы:

SEARCH(S, k) (поиск). Запрос, который по данному множеству S и ключу k возвращает указатель на элемент множества S с ключом k. Если такого элемента в множестве S нет, возвращается NIL.

INSERT(S, x) добавляет к множеству S элемент, на который указывает указатель x (подразумевается, что к этому моменту все поля в записи, на которую указывает x, уже заполнены).

DELETE(S, x) удаляет из множества S элемент, на который указывает указатель x (обратите внимание, что x — указатель, а не ключ).

MINIMUM(S) выдаёт указатель на элемент множества S с наименьшим ключом (считаем, что ключи линейно упорядочены).

MAXIMUM(S) выдаёт указатель на элемент множества S с наибольшим ключом.

SUCCESSOR(S, x) (следующий) возвращает указатель на элемент множества S, непосредственно следующий за элементом x (в смысле линейного порядка на ключах). Если x — наибольший элемент, возвращается NIL.

PREDECESSOR(S, x) (предыдущий) возвращает указатель на элемент, непосредственно предшествующий элементу x (если x — наименьший элемент, возвращается NIL).

Запросы SUCCESSOR и PREDECESSOR часто используются и при работе с множествами, в которых ключи различных элементов могут совпадать. При этом разумная реализация гарантирует, что функции SUCCESSOR и PREDECESSOR обратны, что начав с MINUMUM(S)

и применяя функцию Successor, мы перечислим все элементы множества в неубывающем порядке и т. п.

Стоимость операций над множествами обычно оценивается через размер множеств, к которым они применяются. Например, в главе 14 мы описываем структуру данных, которая позволяет выполнить каждую из перечисленных операций за время $O(\lg n)$, где n — число элементов множества.

Обзор части III

В главах 11–15 мы описываем различные структуры данных, предназначенные для работы с динамическими множествами. С помощью этих структур данных можно разработать эффективные алгоритмы для решения многих различных задач. Кстати, с одной важной структурой данных (кучей) мы уже познакомились в главе 7.

В главе 11 мы разбираем принципы работы с простейшими структурами данных: стеками, очередями, связанными списками и корневыми деревьями. В этой же главе рассказывается, как реализовать записи и указатели с помощью языков программирования, в которых нет соответствующих типов данных. Бóльшая часть материала этой главы составляет стандартный материал начального курса программирования.

В главе 12 мы познакомимся с хеш-таблицами, поддерживающими операции Insert, Delete и Search. В худшем случае поиск в хеш-таблице требует времени $\Theta(n)$, но среднее время, необходимое для выполнения любой из словарных операций с хеш-таблицей, составляет (при некоторых предположениях) лишь $O(1)$. Анализ хеширования использует теорию вероятностей, но для понимания большей части главы знакомство с этой теорией не обязательно.

В главе 13 мы занимаемся деревьями двоичного поиска. Эти деревья поддерживают все перечисленные операции с множествами. В худшем случае стоимость каждой из операций есть $\Theta(n)$, но для случайно построенного дерева математическое ожидание этой стоимости есть $O(\lg n)$. На базе деревьев двоичного поиска строятся многие другие структуры данных.

Глава 14 посвящена красно-чёрным деревьям. Эта разновидность деревьев двоичного поиска гарантированно работает быстро: время выполнения каждой операции в худшем случае есть $O(\lg n)$. Красно-чёрные деревья представляют собой один из вариантов "*сбалансированных*" деревьев поиска; другой вариант (Б-деревья) обсуждается в главе 19. Алгоритмы для работы с красно-чёрными деревьями устроены довольно хитро. Хотя детали можно опустить при первом чтении, интересно в них разобраться.

В главе 15 мы рассматриваем дополнительные структуры на красно-чёрных деревьях, позволяющие выполнять ещё несколько операций (порядковые статистики, операции с деревьями промежутков).

11 Элементарные структуры данных

В этой главе мы рассматриваем несколько простых структур данных для хранения множеств: стеки, очереди, списки, корневые деревья. Мы покажем, как можно реализовать их с помощью указателей или массивов.

11.1 Стеки и очереди

Стеки и очереди — это динамические множества, в которых элемент, удаляемый из множества операцией Delete, не задаётся произвольно, а определяется структурой множества. Именно, из **стека** (stack) можно удалить только тот элемент, который был в него добавлен последним: стек работает по принципу *" последним пришёл — первым ушёл"* (last-in, first-out — сокращенно LIFO). Из **очереди** (queue), напротив, можно удалить только тот элемент, который находился в очереди дольше всего: работает принцип *"первым пришёл — первым ушёл"* (first-in, first-out, сокращенно FIFO). Существует несколько способов эффективно реализовать стеки и очереди. В этом разделе мы расскажем, как реализовать их на базе массива.

Стеки

Операция добавления элемента в стек часто обозначается Push, а операция удаления верхнего элемента из стека часто обозначается Pop (push в данном контексте означает *"запихивать"*, а pop — *"вынимать"*). Стек можно уподобить стопке тарелок, из которой можно взять верхнюю и на которую можно положить новую тарелку. [Другое название стека в русской литературе — *"магазин"* — понятно всякому, кто разбирал автомат Калашникова.]

На рис. 11.1 показано, как можно реализовать стек ёмкостью не более n элементов на базе массива $S[1\,..\,n]$. Наряду с массивом мы храним число $top[S]$, являющееся индексом последнего добавленного в стек элемента. Стек состоит из элементов $S[1\,..\,top[S]]$, где $S[1]$ —

Рисунок 11.1 Реализация стека на базе массива S. Светло-серые клетки заняты элементами стека. (а) Стек S содержит 4 элемента, верхний элемент — число 9. (б) Тот же стек после выполнения операций Push(S, 17) и Push(S, 3). (в) Стек после того, как операция Pop(S) вернула значение 3 (последний добавленный в стек элемент). Хотя число 3 по-прежнему присутствует в массиве S, в стеке его уже нет; на вершине стека — число 17.

нижний элемент стека (*"дно"*) а $S[top[S]]$ — верхний элемент, или вершина стека.

Если $top[S] = 0$, то стек **пуст** (is empty). Если $top[S] = n$, то при попытке добавить элемент происходит **переполнение** (overflow), поскольку размер стека в нашей реализации ограничен числом n. Симметричная ситуация — попытка удалить элемент из пустого стека — по-русски никак не называется (*"недо-полнение"*?), а по-английски называется underflow. В этой главе для простоты мы не будем обращать внимание на возможность переполнения стека.

Операции со стеком (проверка пустоты, добавление элемента, удаление элемента) записываются так:

```
Stack-Empty($S$)
if $top[S] = 0$
    then return TRUE
    else return FALSE
```

```
Push($S, x$)
$top[S] \leftarrow top[S] + 1$
$S[top[S]] \leftarrow x$
```

```
Pop($S$)
if Stack-Empty($S$)
    then error "underflow"
    else $top[S] \leftarrow top[S] - 1$
         return $S[top[S] + 1]$
```

Выполнение операций Push и Pop показано на рис. 11.1. Каждая из трёх операций со стеком выполняется за время $O(1)$.

Очереди

Операцию добавления элемента к очереди мы будем обозначать Enqueue, а операцию удаления элемента из очереди будем обозначать Dequeue. (Как и для стеков, удаляемый из очереди элемент определен однозначно и поэтому является не передаётся процедуре процедуре Dequeue, а возвращается этой процедурой.) Правило здесь такое же, как в живой очереди: первым пришёл — первым обслужен. (И если наши программы правильны, можно не опасаться, что кто-то пройдёт без очереди.)

Другими словами, у очереди есть **голова** (head) и **хвост** (tail). Элемент, добавляемый в очередь, оказывается в её хвосте, как только что подошедший покупатель; элемент, удаляемый из очереди, находится в её голове, как тот покупатель, что отстоял дольше всех.

На рис. 11.2 показано, как можно реализовать очередь, вмещающую не более чем $n-1$ элемент, на базе массива $Q[1\,..\,n]$. Мы храним числа $head[Q]$ — индекс головы очереди, и $tail[Q]$ — индекс свободной ячейки, в которую будет помещён следующий добавляемый к очереди элемент. Очередь состоит из элементов массива, стоящих на местах с номерами $head[Q]$, $head[Q]+1$, …, $tail[Q]-1$ (подразумевается, что массив свёрнут в кольцо: за n следует 1). Если $head[Q] = tail[Q]$, то очередь пуста. Первоначально имеем $head[Q] = tail[Q] = 1$. Если очередь пуста, попытка удалить элемент из неё ведёт к ошибке (underflow); если $head[Q] = tail[Q]+1$, то очередь полностью заполнена, и попытка добавить к ней элемент вызовет переполнение (overflow).

В наших реализациях процедур Enqueue и Dequeue мы игнорируем возможность переполнения или попытки изъятия элемента из пустой очереди (в упражнении 11.1-4 мы попросим вас внести в код соответствующие проверки).

```
Enqueue($Q, x$)
$Q[tail[Q]] \leftarrow x$
if $tail[Q] = length[Q]$
    then $tail[Q] \leftarrow 1$
    else $tail[Q] \leftarrow tail[Q] + 1$
```

```
Dequeue($Q$)
$x \leftarrow Q[head[Q]]$
if $head[Q] = length[Q]$
    then $head[Q] \leftarrow 1$
    else $head[Q] \leftarrow head[Q] + 1$
return $x$
```

Работа процедур Enqueue и Dequeue показана на рис. 11.2. Каждая из этих процедур работает за время $O(1)$.

Рисунок 11.2 Очередь, реализованная на базе массива $Q[1\,..\,n]$. Светло-серые клетки заняты элементами очереди. (а) В очереди находятся 5 элементов (позиции $Q[7\,..\,11]$). (б) Очередь после выполнения процедур Enqueue(Q, 17), Enqueue(Q, 3) и Enqueue(Q, 5). (в) Очередь после выполнения процедуры Dequeue(Q) (которая возвращает значение 15). Новой головой очереди стало число 6.

Упражнения

11.1-1 Следуя образцу рис. 11.1, покажите работу операций Push(S, 4), Push(S, 1), Push(S, 3), Pop(S), Push(S, 8) и Pop(S) на стеке, реализованном с помощью массива $S[1\,..\,6]$. Первоначально стек пуст.

11.1-2 Как на базе одного массива $A[1\,..\,n]$ реализовать два стека суммарной длины не больше n? Операции Push и Pop должны выполняться за время $O(1)$.

11.1-3 Следуя образцу рис. 11.2, покажите работу операций Enqueue(Q, 4), Enqueue(Q, 1), Enqueue(Q, 3), Dequeue(Q), Enqueue(Q, 8) и Dequeue(Q) на очереди, реализованной с помощью массива $Q[1\,..\,5]$. Первоначально очередь пуста.

11.1-4 Перепишите процедуры Enqueue и Dequeue, предусмотрев проверки на случай переполнения или underflow.

11.1-5 Стек позволяет добавлять и удалять элементы только с одного конца. В очередь добавлять элементы можно только с одного конца, а удалять — только с другого. Структура данных, называемая **деком** (deque, от double-ended queue — *"очередь с двумя концами"*), позволяет добавлять и удалять элементы с обоих концов. Реализуйте дек на базе массива таким образом, чтобы операции добавления и удаления элемента с каждого из концов занимали время $O(1)$.

11.1-6 Объясните, как можно реализовать очередь на базе двух стеков. Каково время работы операций Enqueue и Dequeue при такой реализации?

11.1-7 Объясните, как реализовать стек на базе двух очередей. Каково время работы стековых операций?

11.2 Связанные списки

В **связанном списке** (или просто списке; по-английски linked list) элементы линейно упорядочены, но порядок определяется не номерами, как в массиве, а указателями, входящими в состав элементов списка. Списки являются удобным способом реализации динамических множеств, позволяющим реализовать все операции, перечисленные во введении к этой части (хотя и не всегда эффективно).

Если каждый стоящий в очереди запомнит, кто за ним стоит, после чего все в беспорядке рассядутся на лавочке, получится односторонне связанный список; если он запомнит ещё и впереди стоящего, будет двусторонне связанный список.

Другими словами, как показано на рис. 11.3, элемент **двусторонне связанного списка** (doubly linked list) — это запись, содержащая три поля: *key* (ключ) и два указателя *next* (следующий) и *prev* (от previous — предыдущий). Помимо этого, элементы списка могут содержать дополнительные данные. Если x — элемент списка, то $next[x]$ указывает на следующий элемент списка, а $prev[x]$ — на предшествующий. Если $prev[x] = \text{NIL}$, то у элемента x нет предшествующего: это **голова** (head) списка. Если $next[x] = \text{NIL}$, то x — последний элемент списка или, как говорят, его **хвост** (tail).

Прежде чем двигаться по указателям, надо знать хотя бы один элемент списка; мы предполагаем, что для списка L известен указатель $head[L]$ на его голову. Если $head[L] = \text{NIL}$, то список пуст.

В различных ситуациях используются разные виды списков. В **односторонне связанном** (singly linked) списке отсутствуют поля *prev*. В **упорядоченном** (sorted) списке элементы расположены в порядке возрастания ключей, так что у головы списка ключ наименьший, а у хвоста списка — наибольший, в отличие от **неупорядоченного** (unsorted) списка. В **кольцевом списке** (circular list) поле *prev* головы списка указывает на хвост списка, а поле *next* хвоста списка указывает на голову списка.

Если иное не оговорено особо, под списком мы будем понимать неупорядоченный двусторонне связанный список.

Рисунок 11.3 (а) Двусторонне связанный список L содержит числа $1, 4, 9, 16$. Каждый элемент списка — это запись с полями для ключа и указателей на предыдущий и последующий элементы (эти указатели изображены стрелками). В поле *next* у хвоста списка и в поле *prev* у головы списка находится указатель NIL (косая черта на рисунке); $head[L]$ указывает на голову списка. (б) В результате выполнения операции List-Insert(L, x), где $key[x] = 25$, в списке появился новый элемент с ключом 25; он стал новой головой списка, а его поле *next* указывает на бывшую голову — элемент с ключом 9. (в) Вслед за этим была выполнена операция List-Delete(L, x), где x — указатель на элемент с ключом 4.

Поиск в списке

Процедура List-Search(L, k) находит в списке L (с помощью простого линейного поиска) первый элемент, имеющий ключ k. Точнее говоря, она возвращает указатель на этот элемент, или NIL, если элемента с таким ключом в списке нет. Если, например, L — список рис. 11.3а, то вызов List-Search$(L, 4)$ вернёт указатель на третий элемент списка, а вызов List-Search$(L, 7)$ вернёт NIL.

```
List-Search($L, k$)
  $x \leftarrow head[L]$
  while $x \neq$ NIL and $key[x] \neq k$
      do $x \leftarrow next[x]$
  return $x$
```

Поиск в списке из n элементов требует в худшем случае (когда приходится просматривать весь список) $\Theta(n)$ операций.

Добавление элемента в список

Процедура List-Insert добавляет элемент x к списку L, помещая его в голову списка (рис. 11.3б).

```
List-Insert$(L, x)$
$next[x] \leftarrow head[L]$
if $head[L] \neq$ NIL
    then $prev[head[L]] \leftarrow x$
$head[L] \leftarrow x$
$prev[x] \leftarrow$ NIL
```

Процедура List-Insert выполняется за время $O(1)$ (не зависящее от длины списка).

Удаление элемента из списка

Процедура List-Delete удаляет элемент x из списка L, направляя указатели *"в обход"* этого элемента. При этом в качестве аргумента ей передаётся указатель на x. Если задан ключ элемента x, то перед удалением надо найти его указатель с помощью процедуры List-Search.

```
List-Delete$(L, x)$
if $prev[x] \neq$ NIL
    then $next[prev[x]] \leftarrow next[x]$
    else $head[L] \leftarrow next[x]$
if $next[x] \neq$ NIL
    then $prev[next[x]] \leftarrow prev[x]$
```

Удаление элемента из списка проиллюстрировано на рис. 11.3в. Процедура List-Delete работает за время $O(1)$; однако для удаления элемента с заданным ключом его надо сначала найти, что потребует времени $\Theta(n)$.

Фиктивные элементы

Если забыть об особых ситуациях на концах списка, процедуру List-Delete можно записать совсем просто:

```
List-Delete'$(L, x)$
$next[prev[x]] \leftarrow next[x]$
$prev[next[x]] \leftarrow prev[x]$
```

Такие упрощения станут законными, если добавить к списку L фиктивный элемент $nil[L]$, который будет иметь поля *next* и *prev* наравне с прочими элементами списка. Этот элемент (называемый **sentinel** — часовой) не позволит нам выйти за пределы списка. Указатель на него играет роль значения NIL. Замкнём список в кольцо: в поля $next[nil[L]]$ и $prev[nil[L]]$ запишем указатели на голову и хвост списка соответственно, а в поля *prev* у головы списка и *next* у

Рисунок 11.4 Список L, использующий фиктивный элемент $nil[L]$ (тёмно-серый прямоугольник). Вместо $head[L]$ используем $next[nil[L]]$. (а) Пустой список. (б) Список рис. 11.3а (элемент с ключом 9 — голова, 1 — хвост). (в) Тот же список после процедуры List-Insert$'(L, x)$, если $key[x] = 25$. (г) После удаления элемента с ключом 1. Новый хвост имеет ключ 4.

хвоста списка занесём указатели на $nil[L]$ (рис. 11.4). При этом $next[nil[L]]$ — указатель на голову списка, так что атрибут $head[L]$ становится лишним. Пустой список L теперь будет кольцом, в котором $nil[L]$ — единственный элемент.

В процедуре List-Search нужно лишь заменить NIL на $nil[L]$ и $head[L]$ на $next[nil[L]]$:

```
List-Search'$(L, k)$
$x \leftarrow next[nil[L]]$
while $x \neq nil[L]$ and $key[x] \neq k$
    do $x \leftarrow next[x]$
return $x$
```

Для удаления элемента годится процедура List-Delete$'$, приведённая выше. Наконец, добавлять элемент к списку можно так:

```
List-Insert'$(L, x)$
$next[x] \leftarrow next[nil[L]]$
$prev[next[nil[L]]] \leftarrow x$
$next[nil[L]] \leftarrow x$
$prev[x] \leftarrow nil[L]$
```

Пример работы процедур List-Insert$'$ и List-Delete$'$ показан на рис. 11.4.

Использование фиктивных элементов едва ли может улучшить асимптотику времени работы алгоритма, но упрощает программу. Иногда (если использование фиктивных элементов позволяет сократить фрагмент кода, находящийся глубоко внутри цикла), можно ускорить исполнение программы в несколько раз.

Не следует применять фиктивные элементы без нужды. Если алгоритм использует много коротких списков, использование фиктивных элементов может обернуться серьезной дополнительной тратой памяти. В этой книге фиктивные элементы используются только тогда, когда это существенно упрощает программу.

Упражнения

11.2-1 Можно ли добавить элемент в множество, представленное односторонне связанным списком, за время $O(1)$? Тот же вопрос для удаления элемента.

11.2-2 Реализуйте стек на базе односторонне связанного списка. Операции PUSH и POP должны выполняться за время $O(1)$.

11.2-3 Реализуйте очередь на базе односторонне связанного списка. Операции ENQUEUE и DEQUEUE должны выполняться за время $O(1)$.

11.2-4 Реализуйте словарные операции INSERT, DELETE и SEARCH для свёрнутого в кольцо односторонне связанного списка. Каково время работы ваших процедур?

11.2-5 Операция UNION (объединение) получает на входе два непересекающихся множества и возвращает их объединение (сами исходные множества при этом пропадают). Реализуйте эту операцию так, чтобы она работала за время $O(1)$, представляя множества списками подходящего типа.

11.2-6 Напишите процедуру, которая сливает два односторонне связанных упорядоченных списка в один (также упорядоченный), не используя фиктивных элементов. Затем сделайте это, используя фиктивный элемент с ключом ∞ (добавляемый в конец списков). Какая из двух программ проще?

11.2-7 Напишите нерекурсивную процедуру, которая за время $\Theta(n)$ переставляет элементы односторонне связанного списка в обратном порядке. Объём дополнительной (помимо необходимой для хранения исходного списка) памяти должен быть $O(1)$.

11.2-8* Есть способ сэкономить место при реализации двусторонне связанного списка, сжав два указателя *next* и *prev* в одно значение $np[x]$. Будем считать, что все указатели суть k-битные числа и указателю NIL соответствует число нуль. Определим $np[x]$ по формуле $np[x] = next[x] \operatorname{XOR} prev[x]$, где XOR — побитовое сложение по модулю 2 (исключающее ИЛИ). Не забудьте указать, каким образом хранится информация о голове списка. Как реализовать операции SEARCH, INSERT и DELETE? Объясните, как переставить такой список в обратном порядке за время $O(1)$.

Рисунок 11.5 Список рис. 11.3а, представленный с помощью тройки массивов (*key*, *next*, *prev*). Каждому элементу списка соответствует светло-серый столбик. В верхней строчке выписаны порядковые номера, играющие роль указателей; действие указателей показано стрелками. Значение переменной L — указатель на голову списка.

11.3 Реализация указателей и записей с несколькими полями

В языках вроде фортрана не предусмотрено ни указателей, ни элементов, имеющих несколько полей. В таком случае приходится обходиться массивами, используя индекс в массиве как указатель и заменяя массив записей несколькими массивами.

Представление с помощью нескольких массивов

Массив, составленный из записей, можно заменить несколькими массивами (по одному на каждое поле записи). Например, на рис. 11.5 показано, как представить в виде трёх массивов тот же список, что и на рис. 11.3а: в массиве *key* хранятся ключи, а в массивах *next* и *prev* — указатели. Каждому элементу списка соответствует тройка ($key[x]$, $next[x]$, $prev[x]$) для некоторого индекса x. Роль указателя на этот элемент играет число x. В качестве NIL можно использовать число, не являющееся индексом никакого элемента массивов (например, 0 или -1). На рис. 11.3а запись с ключом 4 стоит в списке сразу после записи с ключом 16; и действительно, число 16 стоит в массиве *key* на месте с номером 5, число 4 — на месте номер 2, и имеем $next[5] = 2$, а также $prev[2] = 5$.

В наших программах обозначение типа $key[x]$ может пониматься двояко: и как элемент массива *key* с индексом x, и как поле *key* записи с адресом x (в зависимости от возможностей языка программирования полезно то или другое понимание).

Представление с помощью одного массива

Вместо нескольких массивов можно использовать один, размещая в нём различные поля одного объекта рядом. Так обычно поступает компилятор: если в программе используется массив элементов, ка-

Рисунок 11.6 Тот же список, что на рис. 11.3а и 11.5, реализованный на базе единственного массива A. Каждой записи соответствует тройка идущих подряд элементов массива. Полям *key*, *next* и *prev* соответствуют сдвиги 0, 1 и 2. Указатель на запись — индекс первого из отведённых для неё элементов. Светло-серые записи входят в список; стрелками изображено действие указателей.

ждый из которых имеет несколько полей, то на каждый элемент отводится непрерывный участок памяти, в котором друг за другом размещаются значения полей. Указателем на элемент обычно считают адрес первой ячейки этого участка; адреса полей получаются сдвигом на определённые константы.

При реализации записей на базе массива можно воспользоваться той же стратегией. Рассмотрим один-единственный массив A. Каждая запись будет занимать в нём непрерывный участок $A[j\,..\,k]$. Указатель на запись — это индекс j; каждому полю записи соответствует число из интервала $[0\,..\,k-j]$ — сдвиг. Например, при представлении всё того же списка, что и на рис. 11.3а и 11.5, можно решить, что полям *key*, *next* и *prev* соответствуют сдвиги 0, 1 и 2. Тогда значение $prev[i]$, где i — указатель (= индекс в массиве A), есть не что иное, как $A[i+2]$ (см. рис. 11.6).

Такое представление позволяет хранить в одном массиве записи разных типов (отводя под них участки разной длины), но для наших целей будет достаточно представления в виде нескольких массивов, поскольку большинство структур данных, с которыми мы будем иметь дело, состоят из однотипных записей.

Выделение и освобождение памяти

При добавлении нового элемента в список надо отвести под него место в памяти. Стало быть, необходимо вести учёт использования адресов. В некоторых системах этим ведает специальная подпрограмма — **сборщик мусора** (garbage collector), которая определяет, какие участки памяти более не используются, и возвращает их для повторного использования. Во многих случаях, однако, можно возложить обязанности по выделению и освобождению памяти на саму структуру данных. В этом разделе мы покажем, как это делается; в качестве примера рассмотрим двусторонне связанный список, представленный с помощью нескольких массивов.

Пусть массивы, с помощью которых мы представляем наш спи-

Рисунок 11.7 Работа процедур Allocate-Object и Free-Object. (а) Тот же список, что на рис. 11.5 (светло-серый) и список свободных позиций (тёмно-серый). Структура списка свободных позиций изображена с помощью стрелок. (б) Вот что получится после вызова Allocate-Object() (возвращает значение 4), присваивания $key[4] \leftarrow 25$ и вызова List-Insert$(L, 4)$. Новый список свободных позиций начинается с 8 (таково было значение $next[4]$). (в) Теперь мы вызвали List-Delete$(L, 5)$, а затем Free-Object(5). Позиция 5 — голова нового списка свободных позиций, за ней следует 8.

сок, имеют длину m, и пусть в данный момент в списке содержится $n \leqslant m$ элементов. Остальные $n - m$ мест (позиций) в массиве **свободны** (free).

Мы будем хранить свободные позиции в односторонне связанном списке, называемом **списком свободных позиций** (free list). Этот список использует только массив *next*: именно, $next[i]$ содержит индекс свободной позиции, следующей за свободной позицией i. Голова списка хранится в переменной *free*. Список свободных позиций хранится вперемежку со списком L (рис. 11.7). Обратите внимание, что каждому числу из отрезка $[1; m]$ отвечает элемент либо списка L, либо списка свободных мест.

Список свободных позиций ведёт себя как стек: из всех свободных участков памяти под новую запись выделяется тот, который был освобождён последним. Поэтому для выделения и освобождения памяти можно воспользоваться реализацией стековых операций Push и Pop на базе списка. В приводимых ниже процедурах Allocate-Object (выделить место) и Free-Object (освободить место) подразумевается, что в глобальной переменной $free$ записан индекс первой (в списке) свободной позиции.

Рисунок 11.8 Списки L_1 (светло-серый) и L_2 (тёмно-серый), а также список свободных мест (черный) хранятся в одной тройке массивов.

```
Allocate-Object()
if $free$ = NIL
   then error "Свободного места нет"
   else $x \leftarrow free$
        $free \leftarrow next[x]$
        return $x$
```

```
Free-Object($x$)
$next[x] \leftarrow free$
$free \leftarrow x$
```

Первоначально список свободных позиций содержит n элементов. Если свободного места не осталось, процедура Allocate-Object сообщает об ошибке.

Часто один список свободных позиций обслуживает сразу несколько динамических множеств (рис. 11.8).

Описанные процедуры выделения и освобождения памяти просты и удобны (работают за время $O(1)$). Их можно приспособить для хранения однотипных записей любого вида, отведя одно из полей записи под хранение индекса *next* (в свободных позициях).

Упражнения

11.3-1 Следуя образцу рис. 11.5, изобразите представление последовательности $\langle 13, 4, 8, 19, 5, 11 \rangle$ в виде двусторонне связанного списка, реализованного с помощью трех массивов; по образцу рис. 11.6 изобразите тот же список, реализованный с помощью одного массива.

11.3-2 Напишите процедуры Allocate-Object и Free-Object для случая однотипных записей, хранящихся в одном массиве.

11.3-3 Почему в процедурах Allocate-Object и Free-Object никак не фигурирует поле *prev*?

11.3-4 Часто (например, при страничной организации виртуальной памяти) бывает полезно хранить элементы списка в непрерывном участке памяти. Рассмотрим реализацию списка с помощью нескольких массивов. Перепишите процедуры ALLOCATE-OBJECT и FREE-OBJECT таким образом, чтобы элементы списка занимали позиции $1\,..\,m$, где m — число элементов списка. (Указание: воспользуйтесь реализацией стека на базе массива.)

11.3-5 Пусть двусторонне связанный список L длины m представлен с помощью трех массивов *key*, *prev* и *next* длины n, а процедуры выделения и освобождения места поддерживают *двусторонне* связанный список свободных позиций.

Напишите процедуру COMPACTIFY-LIST (сжатие списка), которая переписывает список L длины m в первые m позиций массивов, сохраняя его структуру (и изменяя список свободных позиций нужным образом). Время работы алгоритма должно быть $\Theta(m)$, размер используемой памяти (сверх занятой массивами) должен быть $O(1)$. Не забудьте доказать правильность своего алгоритма.

11.4 Представление корневых деревьев

Описанные в предыдущем разделе способы представления списков применимы и к другим структурам данных, составленным из однотипных элементов. В этом разделе мы научимся использовать указатели для представления деревьев. Начнём мы с двоичных деревьев, а затем объясним, как представлять деревья с произвольным ветвлением.

Каждая вершина дерева будет записью с несколькими полями. Одно из этих полей содержит ключ, как и в случае со списками. Остальные поля предназначены для хранения дополнительных данных и, главное, указателей на другие вершины. Как конкретно устроены эти поля, зависит от типа дерева.

Двоичные деревья

Как показано на рис. 11.9, при представлении двоичного дерева T мы используем поля p, *left* и *right*, в которых хранятся указатели на родителя, левого и правого ребёнка вершины x соответственно. Если $p[x] = \text{NIL}$, то x — корень; если у x нет левого или правого ребёнка, то $left[x]$ или $right[x]$ есть NIL. С деревом T связан атрибут $root[T]$ — указатель на его корень. Если $root[T] = \text{NIL}$, то дерево T пусто.

Рисунок 11.9 Представление двоичного дерева T. Каждая вершина x включает поля $p[x]$ (сверху), $left[x]$ (внизу слева) и $right[x]$ (внизу справа). Ключи на схеме не показаны.

Корневые деревья с произвольным ветвлением

Если известно, что число детей каждой вершины ограничено сверху константой k, то такое дерево можно реализовать аналогично двоичному дереву, помещая указатели на детей в поля $child_1, child_2, \ldots, child_k$, заменяющие поля *left* и *right*. Если количество детей может быть любым, так делать нельзя: заранее неизвестно, сколько полей (или массивов — при представлении с помощью нескольких массивов) надо выделить.

Проблема может возникнуть и в том случае, если количество детей ограничено заранее известным числом k, но у большинства вершин число детей много меньше k: описанная реализация тратит много памяти зря.

Как же быть? Заметим, что любое дерево можно преобразовать в двоичное. При этом у каждой вершины будет не более двух детей: левый ребёнок останется тем же, но правым ребёнком станет вершина, которая была правым соседом (непосредственно следующим ребёнком того же родителя). Теперь это двоичное дерево можно хранить описанным выше способом.

Опишем схему хранения деревьев с произвольным ветвлением, основанную на этой идее, более подробно. Она называется *"левый ребёнок — правый сосед"* (left-child, right-sibling representation) и показана на рис. 11.10. По-прежнему в каждой вершине хранится указатель p на родителя и атрибут $root[T]$ является указателем на корень дерева. Кроме p, в каждой вершине хранятся ещё два указателя:

1. $left\text{-}child[x]$ указывает на самого левого ребёнка вершины x;

Рисунок 11.10 Представление дерева T по схеме *"левый ребёнок — правый сосед"*. В каждой записи x присутствуют поля $p[x]$ (сверху), *left-child*$[x]$ (внизу слева) и *right-sibling*$[x]$ (внизу справа). Ключи не показаны.

2. *right-sibling*$[x]$ указывает на ближайшего справа соседа вершины x (*"следующего по старшинству брата"*)

Вершина x не имеет детей тогда и только тогда, когда *left-child*$[x] =$ NIL. Если вершина x — крайний правый ребенок своего родителя, то *right-sibling*$[x] =$ NIL.

Другие способы представления деревьев

Иногда встречаются другие способы представления деревьев. Например, в главе 7 при реализации кучи не надо было хранить указателей на родителя и детей, поскольку их номера получались умножением и делением на 2. В главе 22 нам встретятся деревья, по которым двигаются от листьев к корню (и потому указатели на детей или соседей не нужны). Конкретный выбор представления дерева определяется спецификой задачи.

Упражнения

11.4-1 Нарисуйте двоичное дерево, представленное следующим образом:

индекс	*key*	*left*	*right*
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

Корень дерева — в вершине с индексом 6.

11.4-2 Напишите работающую за линейное время рекурсивную процедуру, которая печатает ключи всех вершин данного двоичного дерева.

11.4-3 Как сделать то же самое (что и в предыдущем упражнении), используя нерекурсивную процедуру? (При устранении рекурсии полезен стек.)

11.4-4 Напишите работающую за линейное время процедуру, печатающую ключи всех вершин дерева, представленного по схеме *"левый ребёнок — правый сосед"*.

11.4-5* Напишите работающую за линейное время нерекурсивную процедуру, печатающую ключи всех вершин двоичного дерева, для которой объём используемой памяти (сверх памяти, в которой хранится дерево) есть $O(1)$, и во время работы процедуры дерево не меняется (даже временно).

11.4-6* Придумайте способ хранения дерева с произвольным ветвлением, при котором в каждой вершине хранятся всего два (а не три, как в схеме *"левый ребенок — правый сосед"*) указателя плюс одна булева переменная.

Задачи

11-1 Сравнение разных типов списков

Найдите асимптотику времени работы (в худшем случае) для каждой из перечисленных в начале части III (с. ??) операций, применённой к каждому из указанных типов списков.

	неупорядоченный, односторонне связанный	упорядоченный, односторонне связанный	неупорядоченный, двусторонне связанный	упорядочен двусторонн связанный
Search(L, k)				
Insert(L, x)				
Delete(L, x)				
Successor(L, x)				
Predecessor(L, x)				
Minimum(L)				
Maximum(L)				

11-2 Реализация сливаемых куч на базе списков

Структура данных под названием **сливаемые кучи** (mergeable heaps) хранит набор динамических множеств (куч), и поддерживает следующие операции: Make-Heap (создание пустой кучи), Insert, Minimum, Extract-Min и, наконец, Union (объединение двух куч в одну; две старые кучи пропадают). Для каждого из перечисленных ниже случаев реализуйте (по возможности эффективно) сливаемые кучи на базе списков. Оцените время работы операций через размеры участвующих множеств.

а Списки упорядочены.

б Списки неупорядочены.

в Списки неупорядочены, объединяемые множества не пересекаются друг с другом.

11-3 Поиск в отсортированном сжатом списке

В упражнении 11.3-4 требовалось хранить списки *"компактно"*: n-элементный список должен был занимать первые n позиций массива. Предположим ещё, что все ключи различны и что список упорядочен (иными словами, $key[i] < key[next[i]]$ при $next[i] \neq$ NIL). Оказывается, что в этих предположениях следующий вероятностный алгоритм выполняет поиск в списке за время $o(n)$:

```
Compact-List-Search$(L, k)$
$i \leftarrow head[L]$
$n \leftarrow length[L]$
while $i \neq$ NIL and $key[i] \leqslant k$
    do $j \leftarrow$ Random$(1, n)$
       if $key[i] < key[j]$ and $key[j] < k$
          then $i \leftarrow j$
       $i \leftarrow next[i]$
       if $key[i] = k$
          then return $i$
return NIL
```

Без строк 4–6 это был бы обычный алгоритм с последовательным перебором элементов списка. В строках 4–6 мы пытаемся перескочить на случайно выбранную позицию j. Если $key[i] < key[j] < k$, то при этом мы экономим время, так как не проверяем элементы, лежащие в позициях между i и j. (Благодаря тому, что список занимает непрерывный участок массива, мы можем выбрать в нём случайный элемент.)

а Зачем нужно предполагать, что все ключи различны? Покажите, что для неубывающего списка с (возможно) совпадающими ключами случайные скачки могут не улучшить асимптотику времени поиска.

Как оценить время работы? Представим себе, что на нескольких первых шагах мы выполняем только случайные скачки, а на остальных выполняем линейный поиск. Можно оценить ожидаемое расстояние до искомого элемента после первой фазы — и тем самым длительность второй фазы. Наш алгоритм будет работать не хуже такого усечённого, и остаётся только правильно выбрать длительность первой фазы, чтобы получить оценку получше.

Сделаем это аккуратно. Для каждого $t \geqslant 0$ обозначим через X_t случайную величину, равную расстоянию (измеренному вдоль списка) от позиции i до искомого ключа k после t случайных скачков.

б Покажите, что для каждого $t \geqslant 0$ математическое ожидание времени работы алгоритма Compact-List-Search есть $O(t+E[X_t])$.

в Покажите, что $E(X_t) \leqslant \sum_{r=1}^{n}(1-r/n)^t$. (Указание: воспользуйтесь формулой (6.28)).

г Покажите, что $\sum_{r=0}^{n-1} 5 \leqslant n^{t+1}/(t+1)$.

д Покажите, что $E(X_t) \leqslant n/(t+1)$, и объясните *"на пальцах"*, почему это неравенство должно быть верно.

е Покажите, что математическое ожидание времени работы алгоритма Compact-List-Search есть $O(\sqrt{n})$.

Замечания

Прекрасные справочники по структурам данных — книги Ахо, Хопкрофта и Ульмана [5] и Кнута [121]. Результаты экспериментов по сравнению эффективности различных операций на структурах данных можно найти в Гоннет [90]

Стеки и очереди использовались в математике и делопроизводстве в докомпьютерную эру. Кнут [121] отмечает, что в 1947 году Тьюринг (A. M. Turing) использовал стеки для связи подпрограмм.

Структуры данных, основанные на указателях, также, видимо, относятся к *"фольклору"*. Согласно Кнуту, указатели использовались еще в первых компьютерах с магнитными барабанами. В 1951 году Хоппер (G. M. Hopper) разработал язык A-1, в котором алгебраические формулы представлялись в виде двоичных деревьев. Тот же Кнут указывает, что систематическое использование указателей началось с языка IPL-II, который разработали в 1956 году Ньюэлл, Шоу и Саймон (A. Newell, J. C. Shaw, H. A. Simon). В разработанном теми же авторами в 1957 году языке IPL-III появились в явном виде операции со стеками.

12 Хеш-таблицы

Часто бывают нужны динамические множества, поддерживающие только *"словарные операции"* добавления, поиска и удаления элемента. В этом случае часто применяют так называемое хеширование; соответствующая структура данных называется *"хеш-таблица"* (или *"таблица расстановки"*). В худшем случае поиск в хеш-таблице может занимать столько же времени, сколько поиск в списке ($\Theta(n)$), но на практике хеширование весьма эффективно. При выполнении некоторых естественных условий математическое ожидание времени поиска элемента в хеш-таблице есть $O(1)$.

Хеш-таблицу можно рассматривать как обобщение обычного массива. Если у нас достаточно памяти для массива, число элементов которого равно числу всех возможных ключей, для каждого возможного ключа можно отвести ячейку в этом массиве и тем самым иметь возможность добраться до любой записи за время $O(1)$ (*"прямая адресация"*, см. разд. 12.1). Однако если реальное количество записей значительно меньше, чем количество возможных ключей, то эффективнее применить хеширование: вычислять позицию записи в массиве, исходя из ключа. В разделе 12.2 обсуждаются основные идеи, а в разделе 12.3 — конкретные способы такого вычисления. В этой главе представлено несколько вариантов хеширования.

Мы увидим, что хеширование — эффективный и удобный способ выполнять основные словарные операции (среднее время $O(1)$ при некоторых предположениях).

12.1 Прямая адресация

Прямая адресация применима, если количество возможных ключей невелико. Пусть возможными ключами являются числа из множества $U = \{0, 1, \ldots, m-1\}$ (число m не очень велико). Предположим также, что ключи всех элементов различны.

Для хранения множества мы пользуемся массивом $T[0\,..\,m-1]$, называемым **таблицей с прямой адресацией** (direct-address table). Ка-

Переводы надписей на самой картинке: universe of keys — всевозможные ключи, actual keys — используемые ключи, key — ключ, satellite data — дополнительные данные.

Рисунок 12.1 Реализация динамического множества с помощью таблицы T с прямой адресацией. Множество возможных ключей есть $U = \{0, 1, \ldots, 9\}$. Каждому из этих ключей соответствует своё место в таблице. В позициях таблицы с номерами 2, 3, 5 и 8 (фактически используемые ключи) записаны указатели на элементы множества, а в неиспользуемых позициях таблицы (тёмно-серые) записан NIL.

ждая **позиция**, или **ячейка**, (по-английски slot или position) соответствует определённому ключу из множества U (рис. 12.1: $T[k]$ — место, предназначенное для записи указателя на элемент с ключом k; если элемента с ключом k в таблице нет, то $T[k] =$ NIL).

Реализация словарных операций тривиальна:

```
Direct-Address-Search$(T, k)$
    return $T[k]$
```

```
Direct-Address-Insert$(T, x)$
    $T[key[x]] \leftarrow x$
```

```
Direct-Address-Delete$(T, x)$
    $T[key[x]] \leftarrow$ NIL
```

Каждая из этих операций требует времени $O(1)$.

Иногда можно сэкономить место, записывая в таблицу T не указатели на элементы множества, а сами эти элементы. Можно обойтись и без отдельного поля *"ключ"*: ключом служит индекс в массиве. Впрочем, если мы обходимся без ключей и указателей, то надо иметь способ указать, что данная позиция свободна.

Упражнения

12.1-1 Опишите процедуру для нахождения наибольшего элемента динамического множества, представленного в виде таблицы с прямой адресацией. Каково время работы этой процедуры в худшем случае?

12.1-2 **Битовый вектор** (bit vector) — это массив битов (нулей и единиц). Битовый вектор длины m занимает значительно меньше места, чем массив из m указателей. Как, пользуясь битовым вектором, реализовать динамическое множество, состоящее из различных элементов и не содержащее дополнительных данных? Словарные операции должны выполняться за время $O(1)$.

12.1-3 Как реализовать таблицу с прямой адресацией, в которой ключи различных элементов могут совпадать, а сами элементы могут содержать дополнительные данные? Операции добавления, удаления и поиска должны выполняться за время $O(1)$ (не забудьте, что аргументом процедуры удаления DELETE является указатель на удаляемый элемент, а не ключ).

12.1-4* Предположим, что нам надо реализовать динамическое множество, поддерживающее словарные операции, на базе очень большого массива. Первоначально в массиве может быть записано что-то, не имеющее отношение к нашей задаче, но массив такой большой, что предварительно очищать его непрактично. Как в таких условиях реализовать таблицу с прямой адресацией? Каждая запись должна занимать место размером $O(1)$, операции добавления, удаления и поиска должны выполняться за время $O(1)$, время на инициализацию структуры данных также должно быть равно $O(1)$. (Указание: чтобы иметь возможность узнать, имеет ли данный элемент массива отношение к нашей структуре данных, заведите стек, размер которого равен количеству записей в таблице).

12.2 Хеш-таблицы

Прямая адресация обладает очевидным недостатком: если множество U всевозможных ключей велико, то хранить в памяти массив T размером $|U|$ непрактично, а то и невозможно. Кроме того, если число реально присутствующих в таблице записей мало по сравнению с $|U|$, то много памяти тратится зря.

Если количество записей в таблице существенно меньше, чем количество всевозможных ключей, то хеш-таблица занимает гораздо меньше места, чем таблица с прямой адресацией. Именно, хеш-таблица требует памяти объёмом $\Theta(|K|)$, где K — множество записей, при этом время поиска в хеш-таблице по-прежнему есть $O(1)$ (единственное *"но"* в том, что на сей раз это — оценка в среднем, а не в худшем случае, да и то только при определённых предположениях).

В то время как при прямой адресации элементу с ключом k от-

Переводы надписей на самой картинке: universe of keys — всевозможные ключи, actual keys — используемые ключи

Рисунок 12.2 Использование хеш-функции для отображения ключей в позиции хеш-таблицы. Хеш-значения ключей k_2 и k_5 совпадают — имеет место коллизия.

водится позиция номер k, при хешировании этот элемент записывается в позицию номер $h(k)$ в **хеш-таблице** (hash table) $T[0\,..\,m-1]$, где

$$h\colon U \to \{0, 1, \ldots, m-1\}$$

— некоторая функция, называемая **хеш-функцией** (hash function). Число $h(k)$ называют **хеш-значением** (hash value) ключа k. Идея хеширования показана на рис. 12.2: пользуясь массивом длины m, а не $|U|$, мы экономим память.

Проблема, однако, в том, что хеш-значения двух разных ключей могут совпасть. В таких случаях говорят, что случилась **коллизия**, или **столкновение** (collision). К счастью, эта проблема разрешима: хеш-функциями можно пользоваться и при наличии столкновений.

Хотелось бы выбрать хеш-функцию так, чтобы коллизии были невозможны. Но при $|U| > m$ неизбежно существуют разные ключи, имеющие одно и то же хеш-значение. Так что можно лишь надеяться, что для фактически присутствующих в множестве ключей коллизий будет немного, и быть готовыми обрабатывать те коллизии, которые всё-таки произойдут.

Выбирая хеш-функцию, мы обычно не знаем заранее, какие именно ключи будут храниться. Но на всякий случай разумно сделать сделать хеш-функцию в каком-то смысле *"случайной"*, хорошо перемешивающей ключи по ячейкам (английский глагол “to hash” означает *"мелко порубить, помешивая"*). Разумеется, *"случайная"* хеш-функция должна всё же быть детерминированной в том смысле, что при ее повторных вызовах с одним и тем же аргументом она должна возвращать одно и то же хеш-значение.

В этом разделе мы рассмотрим простейший способ обработки (как говорят, *"разрешения"*) коллизий с помощью цепочек. Другой способ — открытая адресация — рассматривается в разделе 12.4.

Переводы надписей на самой картинке: universe of keys — всевозможные ключи, actual keys — используемые ключи

Рисунок 12.3 Разрешение коллизий с помощью цепочек. В позиции $T[j]$ хранится указатель на список элементов с хеш-значением j. Например, $h(k_1) = h(k_4)$ и $h(k_5) = h(k_2) = h(k_7)$.

Разрешение коллизий с помощью цепочек

Технология **сцепления элементов** (chaining) состоит в том, что элементы множества, которым соответствует одно и то же хеш-значение, связываются в цепочку-список (рис. 12.3). В позиции номер j хранится указатель на голову списка тех элементов, у которых хеш-значение ключа равно j; если таких элементов в множестве нет, в позиции j записан NIL.

Операции добавления, поиска и удаления реализуются легко:

```
Chained-Hash-Insert($T, x$)
    добавить $x$ в голову списка $T[h(key[x])]$

Chained-Hash-Search($T, k$)
    найти элемент с ключом $k$ в списке $T[h(k)]$

Chained-Hash-Delete($T, x$)
    удалить $x$ из списка $T[h(key[x])]$
```

Операция добавления работает в худшем случае за время $O(1)$. Максимальное время работы операции поиска пропорционально длине списка (ниже мы рассмотрим этот вопрос подробнее). Наконец, удаление элемента можно провести за время $O(1)$ — при условии, что списки двусторонне связаны (если списки связаны односторонне, то для удаления элемента x надо предварительно найти его предшественника, для чего необходим поиск по списку; в таком случае стоимость удаления и поиска примерно одинаковы).

Анализ хеширования с цепочками

В этом разделе мы оценим время работы операций для хеширования с цепочками.

Пусть T — хеш-таблица с m позициями, в которую занесено n элементов. **Коэффициентом заполнения** (load factor) таблицы называется число $\alpha = n/m$ (это число может быть и меньше, и больше единицы). Мы будем оценивать стоимость операций в терминах α.

В худшем случае хеширование с цепочками ведет себя отвратительно: если хеш-значения всех n ключей совпадают, то таблица сводится к одному списку длины n, и на поиск будет тратиться то же время $\Theta(n)$, что и на поиск в списке, плюс ещё время на вычисление хеш-функции. Конечно, в такой ситуации хеширование бессмысленно.

Средняя стоимость поиска зависит от того, насколько равномерно хеш-функция распределяет хеш-значения по позициям таблицы. Вопросу о том, как добиваться этой равномерности, посвящён раздел 12.3; пока же будем условно предполагать, что каждый данный элемент может попасть в любую из m позиций таблицы с равной вероятностью и независимо от того, куда попал другой элемент. Мы будем называть это предположение гипотезой *"равномерного хеширования"* (simple uniform hashing).

Будем считать, что для данного ключа k вычисление хеш-значения $h(k)$, шаг по списку и сравнение ключей требует фиксированного времени, так что время поиска элемента с ключом k линейно зависит от количества элементов в списке $T[h(k)]$, которые мы просматриваем в процессе поиска. Будем различать два случая: в первом случае поиск оканчивается неудачей (элемента с ключом k в списке нет), во втором поиск успешен — элемент с требуемым ключом обнаруживается.

Теорема 12.1. *Пусть T — хеш-таблица с цепочками, имеющая коэффициент заполнения α. Предположим, что хеширование равномерно. Тогда при поиске элемента, отсутствующего в таблице, будет просмотрено в среднем α элементов таблицы, а среднее время такого поиска (включая время на вычисление хеш-функции) будет равно $\Theta(1+\alpha)$.*

Доказательство. Поскольку в предположении равномерного хеширования все позиции таблицы для данного ключа равновероятны, среднее время поиска отсутствующего элемента совпадает со средним временем полного просмотра одного из m списков, то есть пропорционально средней длине наших m списков. Эта средняя длина есть $n/m = \alpha$, откуда получаем первое утверждение теоремы; второе утверждение получится, если добавить время $\Theta(1)$ на вычисление хеш-значения. $\square$

Теорема 12.2. *При равномерном хешировании среднее время успешного поиска в хеш-таблице с цепочками есть $\Theta(1+\alpha)$, где α — коэффициент заполнения.*

Доказательство. Хотя формулировка этой теоремы похожа на предыдущую, смысл утверждения несколько иной. В предыдущей теореме мы рассматривали произвольную таблицу с коэффициентом заполнения α и оценивали среднее число действий, необходимых для поиска случайного элемента, равновероятно попадающего во все ячейки таблицы.

В этой теореме так делать нельзя: если мы возьмём произвольную таблицу и, считая все её элементы равновероятными, будем искать среднее время поиска случайно выбранного из них, то оценки вида $\Theta(1+\alpha)$ не получится (контрпример: таблица, в которой все элементы попали в один список)

Формулировка подразумевает двойное усреднение: сначала мы рассматриваем случайно выбранную последовательность элементов, добавляемых в таблицу, причём на каждом шаге все значения ключа равновероятны и шаги независимы, а затем в полученной таблице выбираем элемент для поиска, считая все её элементы равновероятными.

Посмотрим на ситуацию в тот момент, когда таблица уже построена, но случайный элемент для поиска ещё не выбран. Чему равно среднее время поиска, усреднённое по всем n элементам таблицы? Надо сложить позиции всех элементов в своих списках и поделить сумму на n (общее число элементов).

Если представить себе, что при заполнении таблицы элементы дописывались в конец соответствующих списков (см. упр. 12.2-3), то упомянутая сумма по порядку величины равна общему число операций, выполненных при заполнении таблицы (поскольку при добавлении в конец и при поиске выполняется одно и то же количество действий).

Теперь вспомним об усреднении по различным возможностям в процессе построения таблицы. При добавлении в неё i-го элемента математическое ожидание числа действий равно $\Theta(1+(i-1)/m)$ (см. доказательство предыдущей теоремы), и потому математическое ожидание общего числа действий при заполнении таблицы, делённое на n, есть

$$\begin{aligned}\Theta\left(\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{m}\right)\right) &= \Theta\left(1+\frac{1}{nm}\sum_{i=1}^{n}(i-1)\right) = \\ &= \Theta\left(1+\frac{1}{nm}\cdot\frac{(n-1)n}{2}\right) = \\ &= \Theta\left(1+\frac{\alpha}{2}-\frac{1}{2m}\right) = \Theta(1+\alpha).\end{aligned}$$

□

Если количество позиций в хеш-таблице считать пропорциональным числу элементов в таблице, то из доказанных теорем вытекает, что среднее время на поиск (в оптимистических предположениях о распределении вероятностей) есть $O(1)$. В самом деле, если $n = O(m)$, то $\alpha = n/m = O(1)$ и $O(1+\alpha) = O(1)$. Поскольку стоимость добавления в хеш-таблицу с цепочками есть $O(1)$ (даже при добавлении в конец, см. упр. 12.2-3), а стоимость удаления элемента есть $O(1)$ (мы считаем, что списки двусторонне связаны), среднее время выполнения любой словарной операции (в предположении равномерного хеширования) есть $O(1)$.

Упражнения

12.2-1 Пусть h — случайная хеш-функция, сопоставляющая с каждым из n различных ключей $\{k_1, k_2, \ldots, k_n\}$ одну из m позиций в таблице. Каково математическое ожидание числа коллизий (точнее, числа пар (i,j), для которых что $h(k_i) = h(k_j)$)?

12.2-2 Как будет выглядеть хеш-таблица с цепочками после того, как в неё последовательно поместили элементы с ключами $5, 28, 19, 15, 20, 33, 12, 17, 10$ (в указанном порядке)? Число позиций в таблице равно 9, хеш-функция имеет вид $h(k) = k \bmod 9$.

12.2-3 Покажите, что математическое ожидание времени добавления нового элемента (в предположении равномерного хеширования) есть $O(1+\alpha)$, если мы добавляем новый элемент в конец соответствующей цепочки.

12.2-4 Профессор предполагает, что хеширование с цепочками будет гораздо эффективнее, если списки элементов с данным хеш-значением будут упорядоченными. Как этот подход повлияет на стоимость успешного поиска, поиска отсутствующего элемента, добавления, удаления?

12.2-5 Разработайте реализацию хеш-таблицы с цепочками, в которой записи хранятся внутри самой хеш-таблицы (неиспользуемые позиции связываются в список свободных мест). Считайте, что в каждой позиции могут храниться либо флаг и два указателя, либо флаг, указатель и элемент. Все словарные операции, а также операции по выделению и освобождению места, должны выполняться за время $O(1)$. Обязательно ли делать список свободных мест двусторонне связанным?

12.2-6 Пусть общее число возможных ключей (размер множества U) превосходит mn, где m — количество хеш-значений. Покажите, что существует не менее n ключей с одним и тем же хеш-значением, так что в худшем случае поиск в хеш-таблице с цепочками займет время $\Theta(n)$.

12.3 Хеш-функции

В этом разделе мы обсудим, чего мы ждём от хорошей хеш-функции, а затем разберём три способа построения хеш-функций: деление с остатком, умножение и универсальное хеширование.

Какой должна быть хорошая хеш-функция?

Хорошая хеш-функция должна (приближенно) удовлетворять предположениям равномерного хеширования: для очередного ключа все m хеш-значений должны быть равновероятны. Чтобы это предположение имело смысл, фиксируем распределение вероятностей P на множестве U; будем предполагать, что ключи выбираются из U независимо друг от друга, и каждый распределён в соответствии с P. Тогда равномерное хеширование означает, что

$$\sum_{k\colon h(k)=j} P(k) = \frac{1}{m} \quad \text{для } j = 0, 1, \ldots, m-1. \tag{12.1}$$

К сожалению, распределение P обычно неизвестно, так что проверить это невозможно (да и ключи не всегда разумно считать независимыми).

Изредка распределение P бывает известно. Пусть, например, ключи — случайные действительные числа, независимо и равномерно распределённые на интервале $[0; 1)$. В этом случае легко видеть, что хеш-функция $h(k) = \lfloor km \rfloor$ удовлетворяет условию (12.1).

На практике при выборе хеш-функций пользуются различными эвристиками, основанными на специфике задачи. Например, компилятор языка программирования хранит таблицу символов, в которой ключами являются идентификаторы программы. Часто в программе используется несколько похожих идентификаторов (например, `pt` и `pts`). Хорошая хеш-функция будет стараться, чтобы хеш-значения у таких похожих идентификаторов были различны.

Обычно стараются подобрать хеш-функцию таким образом, чтобы её поведение не коррелировало с различными закономерностями, которые могут встретиться в хешируемых данных. Например, описываемый ниже метод деления с остатком состоит в том, что в качестве хеш-значения берётся остаток от деления ключа на некоторое простое число. Если это простое число никак не связано с функцией распределения P, то такой метод даёт хорошие результаты.

Заметим в заключение, что иногда желательно, чтобы хеш-функция удовлетворяла условиям, выходящим за пределы требования равномерного хеширования. Например, можно стараться, чтобы *"близким"* в каком-либо смысле ключам соответствовали *"далёкие"* хеш-значения (это особенно желательно при пользовании описанной в разделе 12.4 линейной последовательностью проб).

Ключи как натуральные числа

Обычно предполагают, что область определения хеш-функции — множество целых неотрицательных чисел. Если ключи не являются натуральными числами, их обычно можно преобразовать к такому виду (хотя числа могут получиться большими). Например, последовательности символов можно интерпретировать как числа, записанные в системе счисления с подходящим основанием: идентификатор `pt` — это пара чисел $(112, 116)$ (таковы ASCII-коды букв `p` и `t`), или же число $(112 \cdot 128) + 116 = 14452$ (в системе счисления по основанию 128). Далее мы всегда будем считать, что ключи — целые неотрицательные числа.

12.3.1 Деление с остатком

Построение хеш-функции методом **деления с остатком** (division method) состоит в том, что ключу k ставится в соответствие остаток от деления k на m, где m — число возможных хеш-значений:

$$h(k) = k \bmod m.$$

Например, если размер хеш-таблицы равен $m = 12$ и ключ равен 100, то хеш-значение равно 4.

При этом некоторых значений m стоит избегать. Например, если $m = 2^p$, то $h(k)$ — это просто p младших битов числа k. Если нет уверенности, что все комбинации младших битов ключа будут встречаться с одинаковой частотой, то степень двойки в качестве числа m не выбирают. Нехорошо также выбирать в качестве m степень десятки, если ключи естественно возникают как десятичные числа: ведь в этом случае окажется, что уже часть цифр ключа полностью определяет хеш-значение. Если ключи естественно возникают как числа в системе счисления с основанием 2^p, то нехорошо брать $m = 2^p - 1$, поскольку при этом одинаковое хеш-значение имеют ключи, отличающиеся лишь перестановкой *"2^p-ичных цифр"*.

Хорошие результаты обычно получаются, если выбрать в качестве m простое число, далеко отстоящее от степеней двойки. Пусть, например, нам надо поместить примерно 2000 записей в хеш-таблицу с цепочками, причем нас не пугает возможный перебор

Переводы надписей: w bits — w битов; extract p bits — выделить p битов. ВНИМАНИЕ: на рисунке надо УБРАТЬ знаки целой части, заменив $\lfloor A \cdot 2^w \rfloor$ на $A \cdot 2^w$!!!!!!!

Рисунок 12.4 Хеширование методом умножения. Ключ k, представленный в виде w-битного числа, умножается на w-битное число $A \cdot 2^w$, где A — константа из интервала $(0; 1)$. У произведения берут младшие w битов, а из этих w битов выделяют p старших. Это и есть хеш-значение $h(k)$.

трёх вариантов при поиске отсутствующего в таблице элемента. Что ж, воспользуемся методом деления с остатком при длине хеш-таблицы $m = 701$. Число 701 простое, $701 \approx 2000/3$, и до степеней двойки от числа 701 тоже далеко. Стало быть, можно выбрать хеш-функцию вида

$$h(k) = k \bmod 701.$$

На всякий случай можно ещё поэкспериментировать с реальными данными на предмет того, насколько равномерно будут распределены их хеш-значения.

12.3.2 Умножение

Построение хеш-функции методом **умножения** (multiplication method) состоит в следующем. Пусть количество хеш-значений равно m. Зафиксируем константу A в интервале $0 < A < 1$, и положим

$$h(k) = \lfloor m(kA \bmod 1) \rfloor,$$

где $kA \bmod 1$ — дробная часть kA.

Достоинство метода умножения в том, что качество хеш-функции мало зависит от выбора m. Обычно в качестве m выбирают степень двойки, поскольку в большинстве компьютеров умножение на такое m реализуется как сдвиг слова. Пусть, например, длина слова в нашем компьютере равна w битам и ключ k помещается в одно слово. Тогда, если $m = 2^p$, то вычисление хеш-функции можно провести так: умножим k на w-битное целое число $A \cdot 2^w$ (мы предполагаем, что это число является целым); получится $2w$-битное число

Метод умножения работает при любом выборе константы A, но некоторые значения A могут быть лучше других. Оптимальный выбор зависит от того, какого рода данные подвергаются хешированию. В книге [123] Кнут обсуждает выбор константы A и приходит к выводу, что значение

$$A \approx (\sqrt{5} - 1)/2 = 0{,}6180339887\ldots \qquad (12.2)$$

является довольно удачным.

В заключение приведём пример: если $k = 123456$, $m = 10000$ и A определено формулой (12.2), то

$$\begin{aligned} h(k) &= \lfloor 10000 \cdot (123456 \cdot 0{,}61803\ldots \bmod 1) \rfloor = \\ &= \lfloor 10000 \cdot (76300{,}0041151\ldots \bmod 1) \rfloor = \\ &= \lfloor 10000 \cdot 0{,}0041151\ldots \rfloor = \\ &= \lfloor 41{,}151\ldots \rfloor = 41. \end{aligned}$$

12.3.3 Универсальное хеширование

Если недоброжелатель будет специально подбирать данные для хеширования, то (зная функцию h) он может устроить так, что все n ключей будут соответствовать одной позиции в таблице, в результате чего время поиска будет равно $\Theta(n)$. Любая фиксированная хеш-функция может быть дискредитирована таким образом. Единственный выход из положения — выбирать хеш-функцию случайным образом, не зависящим от того, какие именно данные вы хешируете. Такой подход называется **универсальным хешированием** (universal hashing). Что бы ни предпринимал ваш недоброжелатель, если он не имеет информации о выбранной хеш-функции, среднее время поиска останется хорошим.

Основная идея универсального хеширования — выбирать хеш-функцию во время исполнения программы случайным образом из некоторого множества. Стало быть, при повторном вызове с теми же входными данными алгоритм будет работать уже по-другому. Как и в случае с алгоритмом быстрой сортировки, рандомизация гарантирует, что нельзя придумать входных данных, на которых алгоритм всегда бы работал медленно (в примере с компилятором и таблицей символов не сможет получиться, что какой-то определённый стиль выбора идентификаторов приводит к замедлению компиляции: вероятность, что компиляция замедлится из-за неудачного хеширования, во-первых, мала, и во-вторых, зависит только от количества идентификаторов, но не от их выбора).

Пусть $\mathcal{H}$ — конечное семейство функций, отображающих данное множество U (множество всевозможных ключей) во множество $\{0, 1, \ldots, m-1\}$ (множество хеш-значений). Это семейство называется **универсальным** (universal), если для любых двух ключей $x, y \in U$ число функций $h \in \mathcal{H}$, для которых $h(x) = h(y)$, равно $|\mathcal{H}|/m$. Иными словами, при случайном выборе хеш-функции вероятность коллизии между двумя данными ключами должна равняться вероятности совпадения двух случайно выбранных хеш-значений (которая равна $1/m$).

Следующая теорема показывает, что универсальное семейство хеш-функций обеспечивает хорошую производительность в среднем.

Теорема 12.3. *Пусть нам необходимо поместить n фиксированных ключей в таблицу размера m, где $m \geqslant n$, и хеш-функция выбирается случайным образом из универсального семейства. Тогда математическое ожидание числа коллизий, в которых участвует данный ключ x, меньше единицы.*

Доказательство. Математическое ожидание числа коллизий данного ключа x с данным ключом y равно $1/m$ по определению универсального семейства. Поскольку всего имеется $n-1$ ключей, отличных от x, математическое ожидание числа коллизий с каким-нибудь из этих ключей равно $(n-1)/m$, что меньше единицы, поскольку $n \leqslant m$. ☐

Как же построить универсальное семейство? Нам поможет в этом элементарная теория чисел. Число m (количество хеш-значений) выберем простым. Будем считать, что каждый ключ представляет собой последовательность $r+1$ *"байтов"* (байт, или символ,— это просто двоичное число с ограниченным числом разрядов; мы будем считать, что максимальное значение байта меньше m). Для каждой последовательности $a = \langle a_0, a_1, \ldots, a_r \rangle$, элементы которой являются вычетами по модулю m (то есть принадлежат множеству $\{0, 1, \ldots, m-1\}$), рассмотрим функцию h_a, заданную формулой

$$h_a(x) = \sum_{i=0}^{r} a_i x_i \bmod m, \tag{12.3}$$

где ключ x есть последовательность байтов $\langle x_0, x_1, \ldots, x_r \rangle$. Положим

$$\mathcal{H} = \bigcup_a \{h_a\}, \tag{12.4}$$

Очевидно, множество $\mathcal{H}$ содержит m^{r+1} элементов.

Теорема 12.4. *Семейство функций $\mathcal{H}$, определённое по формулам* (12.3) *и* (12.4)*, является универсальным семейством хеш-функций.*

Доказательство. Пусть $x = \langle x_0, x_1, \ldots, x_r \rangle$ и $y = \langle y_0, y_1, \ldots, y_r \rangle$ — два различных ключа; не ограничивая общности, можно считать, что $x_0 \neq y_0$. Если $a = \langle a_0, a_1, \ldots, a_r \rangle$, то $h_a(x) = h_a(y)$ тогда и только тогда, когда

$$a_0(x_0 - y_0) \equiv -\sum_{j=1}^{r} a_i(x_i - y_i) \pmod{m}.$$

Поскольку $x_0 - y_0 \not\equiv 0 \pmod{m}$, для каждой последовательности $\langle a_1, \ldots, a_r \rangle$ существует и единственно значение a_0, при котором это равенство выполнено (раздел 33.4). Количество таких последовательностей равно m^r, и таково же, стало быть, количество функций из $\mathcal{H}$, не различающих ключи x и y. Поскольку $m^r = |\mathcal{H}|/m$, всё доказано.

[Короче можно сказать так: ненулевой линейный функционал $h \mapsto h(x-y)$ с равной вероятностью принимает любое из m своих значений, в том числе 0.] ☐

Упражнения

12.3-1 Пусть в связанном списке каждый элемент хранится вместе с его ключом k и соответствующим хеш-значением $h(k)$. Ключ представляет собой длинную последовательность символов. Как можно упростить поиск в этом списке элемента с данным ключом?

12.3-2 Предположим, что ключами являются последовательности символов, которые мы рассматриваем как числа, записанные в системе счисления с основанием 128. Число m помещается в 32-битном слове с запасом, но числа, соответствующие ключам, уже не помещаются, поскольку ключи содержат много десятков символов. Как вычислить хеш-функцию, построенную методом деления? (Нет необходимости реализовывать арифметические операции с длинными числами — достаточно дополнительной памяти постоянного объёма.)

12.3-3 Пусть ключи представляют собой последовательности p-битных символов, рассматриваемые как числа в 2^p-ичной системе счисления, а в качестве хеш-функции выбран остаток при делении на $m = 2^p - 1$. Покажите, что двум ключам, отличающимся только порядком символов, соответствует одно и то же хеш-значение. Приведите пример приложения, в котором применение такой хеш-функции было бы нежелательно.

12.3-4 Пусть размер хеш-таблицы равен $m = 1000$, а хеш-функция имеет вид $h(k) = \lfloor m(kA \bmod 1) \rfloor$, где $A = (\sqrt{5} - 1)/2$. В какие позиции попадут ключи 61, 62, 63, 64 и 65?

12.3-5 Удалим из семейства $\mathcal{H}$, определённого по формулам (12.3) и (12.4), те функции h_a, в которых хотя бы одно из a_i равна нулю. Покажите, что получившееся семейство хеш-функций универсальным не будет.

12.4 Открытая адресация

В отличие от хеширования с цепочками, при **открытой адресации** (open addressing) никаких списков нет, а все записи хранятся в самой хеш-таблице: каждая ячейка таблицы содержит либо элемент динамического множества, либо NIL. Поиск заключается в том, что мы определённым образом просматриваем элементы таблицы, пока не найдём то, что ищем, или не удостоверимся, что элемента с таким ключом в таблице нет. Тем самым число хранимых элементов не может быть больше размера таблицы: коэффициент заполнения не больше 1.

Конечно, и при хешировании с цепочками можно использовать свободные места в хеш-таблице для хранения списков (упражнение 12.2-5), но при открытой адресации указатели вообще не используются: последовательность просматриваемых ячеек *вычисляется*. За счет экономии памяти на указателях можно увеличить количество позиций в таблице, что уменьшает число коллизий и сокращает поиск.

Чтобы добавить новый элемент в таблицу с открытой адресацией, ячейки которой занумерованы целыми числами от 0 до $m-1$, мы просматриваем её, пока не найдем свободное место. Если всякий раз просматривать ячейки подряд $(0, 1, \ldots, m-1)$, потребуется время $\Theta(n)$, но суть в том, что порядок просмотра таблицы зависит от ключа! Иными словами, мы добавляем к хеш-функции второй аргумент — номер попытки (нумерацию начинаем с нуля), так что хеш-функция имеет вид

$$h\colon U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

(U — множество ключей). **Последовательность испробованных мест** (probe sequence), или (короче) **последовательность проб** для данного ключа k имеет вид

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle;$$

функция h должна быть такой, чтобы каждое из чисел от 0 до $m-1$ встретилось в этой последовательности ровно один раз (для каждого ключа все позиции таблицы должны быть доступны). Ниже приводится текст процедуры добавления в таблицу T с открытой адресацией; в нем подразумевается, что записи не содержат дополнительной информации, кроме ключа. Если ячейка таблицы пуста, в ней записан NIL (фиксированное значение, отличное от всех ключей).

```
HASH-INSERT($T, k$)
  $i \leftarrow 0$
  repeat $j \leftarrow h(k, i)$
         if $T[j] =$ NIL
            then $T[j] \leftarrow k$
                 return $j$
            else $i \leftarrow i + 1$
    until $i = m$
  error "переполнение хеш-таблицы"
```

При поиске элемента с ключом k в таблице с открытой адресацией ячейки таблицы просматриваются в том же порядке, что и при добавлении в нее элемента с ключом k. Если при этом мы натыкаемся на ячейку, в которой записан NIL, то можно быть уверенным, что искомого элемента в таблице нет (иначе он был бы занесён в эту ячейку). (Внимание: мы предполагаем, что никакие элементы из таблицы не удаляются!)

Вот текст процедуры поиска HASH-SEARCH (если элемент с ключом k содержится в таблице T в позиции j, процедура возвращает j, в противном случае она возвращает NIL).

```
HASH-SEARCH($T, k$)
  $i \leftarrow 0$
  repeat $j \leftarrow h(k, i)$
         if $T[j] = k$
            then return $j$
         $i \leftarrow i + 1$
    until $T[j] =$ NIL или $i = m$
  return NIL
```

Удалить элемент из таблицы с открытой адресацией не так просто. Если просто записать на его место NIL, то в дальнейшем мы не сможем найти те элементы, в момент добавления которых в таблицу это место было занято (и из-за этого был выбран более далёкий элемент в последовательности испробованных мест). Возможное решение — записывать на место удалённого элемента не NIL, а специальное значение DELETED (*"удалён"*), и при добавлении рассматривать ячейку с записью DELETED как свободную, а при поиске — как занятую (и продолжать поиск). Недостаток этого подхода в том, что время поиска может оказаться большим даже при низком коэффициенте заполнения. Поэтому, если требуется удалять записи из хеш-таблицы, предпочтение обычно отдают хешированию с цепочками.

В нашем анализе открытой адресации мы будем исходить из предположения, что хеширование **равномерно** (uniform) в том смысле, что все $m!$ перестановок множества $\{0, 1, \ldots, m-1\}$ равновероятны. На практике это вряд ли так, хотя бы по той причине, что для этого необходимо, чтобы число возможных ключей было как минимум $\leqslant m!$, где m — число хеш-значений. Поэтому обычно пользуются более или менее удачными суррогатами, вроде описываемого ниже двойного хеширования.

Обычно применяют такие три способа вычисления последовательности испробованных мест: линейный, квадратичный и двойное хеширование. В каждом из этих способов последовательность $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ будет перестановкой множества $\{0, 1, \ldots, m-1\}$ при любом значении ключа k, но ни один из этих способов не является равномерным по той причине, что они дают не более m^2 перестановок из $m!$ возможных. Больше всего разных перестановок получается при двойном хешировании; не удивительно, что и на практике этот способ дает лучшие результаты.

Линейная последовательность проб

Пусть $h'\colon U \to \{0, 1, \ldots, m-1\}$ — *"обычная"* хеш-функция. Функция, определяющая **линейную последовательность проб** (linear probing), задаётся формулой

$$h(k, i) = (h'(k) + i) \bmod m.$$

Иными словами, при работе с ключом k начинают с ячейки $T[h'(k)]$, а затем перебирают ячейки таблицы подряд: $T[h'(k)+1], T[h'(k)+2], \ldots$ (после $T[m-1]$ переходят к $T[0]$). Поскольку последовательность проб полностью определяется первой ячейкой, реально используется всего лишь m различных последовательностей.

Открытую адресацию с линейной последовательностью проб легко реализовать, но у этого метода есть один недостаток: он может привести к образованию **кластеров**, то есть длинных последовательностей занятых ячеек, идущих подряд (по-английски это явление называется primary clustering). Это удлиняет поиск; в самом деле, если в таблице из m ячеек все ячейки с чётными номерами заняты, а ячейки с нечётными номерами свободны, то среднее число проб при поиске элемента, отсутствующего в таблице, есть 1,5; Если, однако, те же $m/2$ занятых ячеек идут подряд, то среднее число проб примерно равно $m/8 = n/4$ (n — число занятых мест в таблице). Тенденция к образованию кластеров объясняется просто: если i заполненных ячеек идут подряд, вероятность того, что при очередной вставке в таблицу будет использована ячейка, следующая непосредственно за ними, есть $(i+1)/m$, в то время как для свободной ячейки, предшественница которой также свободна, вероятность быть использованной равна всего лишь $1/m$. Всё вышеизложенное показывает, что линейная последовательность проб довольно далека от равномерного хеширования.

Квадратичная последовательность проб

Функция, определяющая **квадратичную последовательность проб** (quadratic probing), задаётся формулой

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m, \tag{12.5}$$

где по-прежнему h' — обычная хеш-функция, а c_1 и $c_2 \neq 0$ — некоторые константы. Пробы начинаются с ячейки номер $T[h'(k)]$, как и при линейном методе, но дальше ячейки просматриваются не подряд: номер пробуемой ячейки квадратично зависит от номера попытки. Этот метод работает значительно лучше, чем линейный, но если мы хотим, чтобы при просмотре хеш-таблицы использовались все ячейки, значения m, c_1 и c_2 нельзя выбирать как попало; один из способов выбора описан в задаче 12-4. Как и при линейном методе, вся последовательность проб определяется своим первым членом, так что опять получается всего m различных перестановок. Тенденции к образованию кластеров больше нет, но аналогичный эффект проявляется в (более мягкой) форме **образования вторичных кластеров** (secondary clustering).

Двойное хеширование

Двойное хеширование (double hashing) — один из лучших методов открытой адресации. Перестановки индексов, возникающие при двойном хешировании, обладают многими свойствами, присущими равномерному хешированию. При двойном хешировании функция h имеет вид

$$h(k,i) = (h_1(k) + i h_2(k)) \bmod m,$$

где h_1 и h_2 — обычные хеш-функции. Иными словами, последовательность проб при работе с ключом k представляет собой арифметическую прогрессию (по модулю m) с первым членом $h_1(k)$ и шагом $h_2(k)$. Пример двойного хеширования приведен на рис. 12.5.

Чтобы последовательность испробованных мест покрыла всю таблицу, значение $h_2(k)$ должно быть взаимно простым с m (если наибольший общий делитель $h_2(k)$ и m есть d, то арифметическая прогрессия по модулю m с разностью $h_2(k)$ займёт долю $1/d$ в таблице; см. главу 33). Простой способ добиться выполнения этого условия — выбрать в качестве m степень двойки, а функцию h_2 взять такую, чтобы она принимала только нечётные значения. Другой вариант: m — простое число, значения h_2 — целые положительные числа, меньшие m. Например, для простого m можно положить

$$h_1(k) = k \bmod m,$$
$$h_2(k) = 1 + (k \bmod m'),$$

Рисунок 12.5 Добавление элемента в таблицу с открытой адресацией при двойном хешировании. В нашем случае $m = 13$, $h_1(k) = k \bmod 13$, $h_2(k) = 1 + (k \bmod 11)$. Если $k = 14$, то последовательность проб будет такая: 1 и 5 заняты, 9 свободно, помещаем туда.

где m' чуть меньше, чем m (например, $m' = m - 1$ или $m - 2$). Если, например, $m = 701$, $m' = 700$ и $k = 123456$, то $h_1(k) = 80$ и $h_2(k) = 257$. Стало быть, последовательность проб начинается с позиции номер 80 и идёт далее с шагом 257, пока вся таблица не будет просмотрена (или не будет найдено нужное место).

В отличие от линейного и квадратичного методов, при двойном хешировании можно получить (при правильном выборе h_1 и h_2) не m, а $\Theta(m^2)$ различных перестановок, поскольку каждой паре $(h_1(k), h_2(k))$ соответствует своя последовательность проб. Благодаря этому производительность двойного хеширования близка к той, что получилась бы при настоящем равномерном хешировании.

Анализ хеширования с открытой адресацией

Так же, как и при анализе хеширования с цепочками, при анализе открытой адресации мы будем оценивать стоимость операций в терминах коэффициента заполнения $\alpha = n/m$ (n — число записей, m — размер таблицы). Поскольку при открытой адресации каждой ячейке соответствует не более одной записи, $\alpha \leqslant 1$.

Мы будем исходить из предположения о равномерности хеширования. В этой идеализированной схеме предполагается следующее: мы выбираем ключи случайным образом, причём все $m!$ возможных последовательностей проб равновероятны. Поскольку эта идеализированная схема далека от реальности, доказываемые ниже результаты следует рассматривать не как математические теоремы, описывающие работу реальных алгоритмов открытой адресации, а как эвристические оценки.

Начнём с того, что оценим время на поиск элемента, отсутствующего в таблице.

Теорема 12.5. *Математическое ожидание числа проб при поиске в таблице с открытой адресацией отсутствующего в ней элемента не превосходит* $1/(1-\alpha)$ *(хеширование предполагается равномерным, через* $\alpha < 1$ *обозначен коэффициент заполнения).*

Доказательство. Мы предполагаем, что таблица фиксирована, а искомый элемент выбирается случайно, причём все возможные последовательности проб равновероятны. Нас интересует математическое ожидание числа попыток, необходимых для обнаружения свободной ячейки, то есть сумма

$$1+\sum_{i=0}^{\infty} ip_i. \tag{12.6}$$

где p_i — вероятность того, что мы встретим ровно i занятых ячеек.

Каждая новая проба выбирается равномерно среди оставшихся не испробованными ячеек; если разрешить пробовать повторно уже испробованные ячейки, то часть проб пропадёт зря и математическое ожидание только увеличится. Но для этого нового варианта мы уже вычисляли математическое ожидание (раздел 6.4, геометрическое распределение), и оно равно

$$1/(1-\alpha) \tag{12.7}$$

поскольку вероятность успеха для каждой пробы равна $1-\alpha$. □

Если коэффициент заполнения отделён от единицы, теорема 12.5 предсказывает, что поиск отсутствующего элемента будет в среднем проходить за время $O(1)$. Например, если таблица заполнена наполовину, то среднее число проб будет не больше $1/(1-0{,}5)=2$, а если на 90%, то не больше $1/(1-0{,}9)=10$.

Из теоремы 12.5 немедленно получается и оценка на стоимость операции добавления к таблице:

Следствие 12.6. В предположении равномерного хеширования математическое ожидание числа проб при добавлении нового элемента в таблицу с открытой адресацией не превосходит $1/(1-\alpha)$, где $\alpha < 1$ — коэффициент заполнения.

Доказательство. При добавлении в таблицу отсутствующего в ней элемента происходят те же пробы, что при поиске отсутствующего в в ней элемента. □

Оценить среднее время успешного поиска немногим сложнее.

Теорема 12.7. *Рассмотрим таблицу с открытой адресацией, коэффициент заполнения которой равен $\alpha < 1$. Пусть хеширование равномерно. Тогда математическое ожидание числа проб при успешном поиске элемента в таблице не превосходит*

$$\frac{1}{\alpha}\ln\frac{1}{1-\alpha},$$

если считать, что ключ для успешного поиска в таблице выбирается случайным образом и все такие выборы равновероятны.

Доказательство. Уточним, как производится усреднение: сначала мы заполняем таблицу независимо выбираемыми ключами, причём для каждого из них выполняется предположение равномерного хеширования. Затем мы усредняем по всем элементам таблицы время их поиска.

Заметим, что при успешном поиске ключа k мы делаем те же самые пробы, которые производились при помещении ключа k в таблицу. Тем самым среднее число проб при поиске (усреднение по элементам) равно общему числу проб при добавлении, делённому на число элементов в таблице, которое мы обозначаем n. Математическое ожидание общего числа проб при добавлении равно сумме математических ожиданий для каждого отдельного шага. К моменту добавления $(i{+}1)$-го элемента в таблице заполнено i позиций, коэффициент заполнения равен i/m (m — число мест в таблице), и математическое ожидание не больше $1/(1-i/m) = m/(m-i)$. Поэтому сумма математических ожиданий не превосходит

$$\frac{m}{m}+\frac{m}{m-1}+\frac{m}{m-2}+\ldots+\frac{m}{m-n+1}.$$

Эта сумма равна $m\cdot(1/(m-n+1)+\ldots+1/(m-1)+1/m)$ и оценивается сверху с помощью интеграла (3.10):

$$m\cdot\int\limits_{m-n}^{m}\frac{1}{t}\,dt = m\ln(m/(m-n)).$$

Вспоминая, что общее число операций надо поделить на n, получаем оценку $(m/n)\ln(m/(m-n)) = (1/\alpha)\ln(1/(1-\alpha))$. □

Если, например, таблица заполнена наполовину, то среднее число проб для успешного поиска не превосходит 1,387, а если на 90%, то 2,559.

Упражнения

12.4-1 Выполните добавление ключей $10, 22, 31, 4, 15, 28, 17, 88, 59$ (в указанном порядке) в хеш-таблицу с открытой адресацией размера $m = 11$. Для вычисления последовательности проб используется линейный метод с $h'(k) = k \bmod m$. Выполните то же задание, если используется квадратичный метод с той же h', $c_1 = 1$, $c_2 = 3$, а также для двойного хеширования с $h_1 = h'$ и $h_2(k) = 1 + (k \bmod (m-1))$.

12.4-2 Напишите процедуру Hash-Delete для удаления элемента из таблицы с открытой адресацией, реализующую описанную схему (со значением DELETED), и перепишите соответствующим образом процедуры Hash-Insert и Hash-Search.

12.4-3* Покажите, что при двойном хешировании $(h(k,i) = (h_1(k) + ih_2(k)) \bmod m)$ последовательность проб, соответствующая ключу k, является перестановкой множества $\{0, 1, \ldots, m-1\}$ тогда и только тогда, когда $h_2(k)$ взаимно просто с m. (Указание: см. главу 33.)

12.4-4 Найдите численные значения верхних оценок теорем 12.5 и 12.7 для поиске присутствующего и отсутствующего элементов для коэффициентов заполнения 1/2, 3/4 и 7/8.

12.4-5* Предположим, что мы помещаем n записей в таблицу с открытой адресацией; размер таблицы равен m, хеширование равномерно, ключи записей выбираются случайным образом. Обозначим через $p(n,m)$ вероятность того, что при этом не произойдет коллизий. Покажите, что $p(n,m) \leqslant e^{-n(n-1)/2m}$. (Указание: см. неравенство (2.7).) Эта величина очень мала, когда n заметно больше $\sqrt{n}$.

12.4-6* Частичные суммы гармонического ряда можно оценить так:

$$H_n = \ln n + \gamma + \frac{\varepsilon}{2n}, \tag{12.8}$$

где $\gamma = 0{,}5772156649\ldots$ — так называемая **постоянная Эйлера** (Euler's constant) и $0 < \varepsilon < 1$ (доказательство см. в книге Кнута [121]). Как использовать это для оценки куска гармонического ряда в доказательстве теоремы 12.7?

12.4-7* Найдите ненулевое значение коэффициента заполнения α, при котором оценка среднего числа проб при поиске отсутствующего элемента (теорема 12.5) вдвое превосходит оценку среднего числа проб при успешном поиске (теорема 12.7).

Задачи

12-1 Наибольшее число проб при добавлении элемента

Рассмотрим хеш-таблицу с открытой адресацией размера m, в которую один за другим помещаются n ключей, причём $n \leqslant m/2$. Предположим, что хеширование равномерно.

а. Покажите, что для любого $i = 1, 2, \ldots, n$ вероятность того, что при добавлении i-го ключа в таблицу произошло более k проб, не превосходит 2^{-k}.

б. Покажите, что для любого $i = 1, 2, \ldots, n$ вероятность того, что при добавлении i-го ключа в таблицу произошло более $2 \lg n$ проб, не превосходит $1/n^2$.

Пусть X — случайная величина, равная максимальному числу проб при добавлении элементов с номерами $1, 2, \ldots, n$.

в. Покажите, что $\mathsf{P}\{X > 2 \lg n\} \leqslant 1/n$.

г. Покажите, что математическое ожидание величины X (наибольшего числа проб) есть $O(\lg n)$.

12-2 Поиск в неизменяющемся множестве

Пусть мы работаем с множеством из n элементов, в котором ключи являются числами, а единственная операция, которую надо поддерживать, — это поиск (элементы не добавляются и не удаляются). Требуется реализовать поиск максимально эффективно (до того, как начнут поступать запросы на поиск, множество можно предварительно обработать, и время на такую обработку не ограничено).

а. Покажите, что поиск можно реализовать таким образом, чтобы в худшем случае он занимал время $O(\lg n)$, а дополнительная память (сверх той, в которой хранится само множество) не использовалась.

б. Пусть мы решили записать элементы нашего множества в хеш-таблицу с открытой адресацией, состоящую из m ячеек. Предположим, что хеширование равномерно. При каком минимальном объёме $m - n$ дополнительной памяти средняя стоимость операции поиска элемента, отсутствующего в множестве, будет не хуже, чем в пункте (а)? В качестве ответа приведите асимптотическую оценку $m - n$ через n.

12-3 Длины цепочек при хешировании

Рассмотрим хеш-таблицу с n ячейками, в которой коллизии разрешаются с помощью цепочек. Хеширование равномерно: каждый новый ключ имеет равные шансы попасть во все ячейки независимо от предыдущих. Пусть M — максимальная длина цепочек после добавления n ключей. Ваша задача — доказать, что математическое ожидание M есть $O(\lg n/\lg\lg n)$.

а. Фиксируем некоторое хеш-значение. Пусть Q_k — вероятность того, ему соответствует k различных ключей. Покажите, что

$$Q_k = C_n^k \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k}.$$

б. Пусть P_k — вероятность того, что максимальная длина цепочки равна k. Покажите, что $P_k \leqslant nQ_k$.

в. Выведите из формулы Стирлинга (2.11), что $Q_k < e^k/k^k$.

г. Покажите, что существует такая константа $c > 1$, что $Q_k < 1/n^3$ при $k \geqslant c\lg n/\lg\lg n$. Заключите отсюда, что $P_k < 1/n^2$ при таких k.

д. Покажите, что математическое ожидание величины M не превосходит

$$\mathsf{P}\left\{M > \frac{c\lg n}{\lg\lg n}\right\} \cdot n + \mathsf{P}\left\{M \leqslant \frac{c\lg n}{\lg\lg n}\right\} \cdot \frac{c\lg n}{\lg\lg n},$$

и выведите отсюда, что это математическое ожидание есть $O(\lg n/\lg\lg n)$.

12-4 Пример квадратичной последовательности проб

Пусть нам надо поместить запись с ключом k в хеш-таблицу, ячейки которой пронумерованы числами $0, 1, \ldots, m-1$. У нас есть хеш-функция h, отображающая множество ключей в множество $\{0, 1, \ldots, m-1\}$. Будем действовать так:

1. Находим $i \leftarrow h(k)$ и полагаем $j \leftarrow 0$.
2. Проверяем позицию номер i. Если она свободна, заносим туда запись и на этом останавливаемся.
3. Полагаем $j \leftarrow (j+1) \bmod m$ и $i \leftarrow (i+j) \bmod m$ и возвращаемся к шагу 2.

а. Покажите, что описанный алгоритм — частный случай *"квадратичного метода"* для подходящих значений c_1 и c_2.

Предположим, что m является степенью двойки.

б. Покажите, что в худшем случае будет просмотрена вся таблица.

12-5 k-универсальное хеширование

Пусть $\mathcal{H}$ — семейство хеш-функций, отображающих множество возможных ключей U в $\{0, 1, \ldots, m-1\}$. Будем говорить, что $\mathcal{H}$ является **k-универсальным**, если для любой последовательности k различных ключей $\langle x_1, \ldots, x_k\rangle$ случайная величина $\langle h(x_1), \ldots, h(x_k)\rangle$ (где h — случайный элемент $\mathcal{H}$) принимает все m^k своих возможных значений с равными вероятностями.

а. Покажите, что всякое 2-универсальное семейство универсально.

б. Покажите, что семейство $\mathcal{H}$, описанное в разделе 12.3.3, не является 2-универсальным.

в. Расширим семейство $\mathcal{H}$ из разд. 12.3.3 и рассмотрим всевозможные функции вида

$$h_{a,b}(x) = h_a(x) + b \bmod m,$$

где b — некоторый вычет по модулю m. Покажите, что полученное семейство будет 2-универсальным.

Замечания

Алгоритмы хеширования прекрасно изложены у Кнута [123] и Гоннета [90]. Согласно Кнуту, хеш-таблицы и метод цепочек были изобретены Луном (H. P. Luhn) в 1953 году. Примерно в то же время Амдаль (G. M. Amdahl) изобрёл открытую адресацию.

13 Двоичные деревья поиска

Деревья поиска (search trees) позволяют выполнять следующие операции с динамическими множествами: SEARCH (поиск), MINIMUM (минимум), MAXIMUM (максимум), PREDECESSOR (предыдущий), SUCCESSOR (следующий), INSERT (вставить) и DELETE (удалить). Таким образом, дерево поиска может быть использовано и как словарь, и как очередь с приоритетами.

Время выполнения основных операций пропорционально высоте дерева. Если двоичное дерево *"плотно заполнено"* (все его уровни имеют максимально возможное число вершин), то его высота (и время выполнения операций) пропорциональны логарифму числа вершин. Напротив, если дерево представляет собой линейную цепочку из n вершин, это время вырастает до $\Theta(n)$. В разделе 13.4 мы увидим, что высота случайного двоичного дерева поиска есть $O(\lg n)$, так что в этом случае время выполнения основных операций есть $\Theta(\lg n)$.

Конечно, возникающие на практике двоичные деревья поиска могут быть далеки от случайных. Однако, приняв специальные меры по балансировке деревьев, мы можем гарантировать, что высота деревьев с n вершинами будет $O(\log n)$. В главе 14 рассмотрен один из подходов такого рода (красно-чёрные деревья). В главе 19 рассматриваются Б-деревья, которые особенно удобны для данных, хранящихся во вторичной памяти с произвольным доступом (на диске).

В этой главе мы рассмотрим основные операции с двоичными деревьями поиска и покажем, как напечатать элементы дерева в неубывающем порядке, как искать заданный элемент, как найти максимальный или минимальный элемент, как найти элемент, следующий за данным и предшествующий данному, и, наконец, как добавить или удалить элемент. Напомним, что определение дерева и основные свойства деревьев приводятся в главе 5.

Рисунок 13.1 Двоичные деревья поиска. Левое поддерево произвольной вершины x содержит ключи, не превосходящие $key[x]$, правое — не меньшие $key[x]$. Разные двоичные деревья поиска могут представлять одно и то же множество. Время выполнения (в худшем случае) большинства операций пропорционально высоте дерева. (а) Двоичное дерево поиска высоты 2 с 6 вершинами. (б) Менее эффективное дерево высоты 4, содержащее те же ключи.

13.1 Что такое двоичное дерево поиска?

В **двоичном дереве поиска** (binary search tree; пример приведён на рис. 13.1) каждая вершина может иметь (или не иметь) левого и правого ребёнка; каждая вершина, кроме корня, имеет родителя. При представлении с использованием указателей мы храним для каждой вершины дерева, помимо значения ключа *key* и дополнительных данных, также и указатели *left*, *right* и *p* (левый ребёнок, правый ребёнок, родитель). Если ребёнка (или родителя — для корня) нет, соответствующее поле содержит NIL.

Ключи в двоичном дереве поиска хранятся с соблюдением **свойства упорядоченности** (binary-search-tree property):

> Пусть x — произвольная вершина двоичного дерева поиска. Если вершина y находится в левом поддереве вершины x, то $key[y] \leqslant key[x]$. Если y находится в правом поддереве x, то $key[y] \geqslant key[x]$.

Так, на рис. 13.1(а) в корне дерева хранится ключ 5, ключи 2, 3 и 5 левом поддереве корня не превосходят 5, а ключи 7 и 8 в правом — не меньше 5. То же самое верно для всех вершин дерева. Например, ключ 3 на рис. 13.1(а) не меньше ключа 2 в левом поддереве и не больше ключа 5 в правом.

Свойство упорядоченности позволяет напечатать все ключи в неубывающем порядке с помощью простого рекурсивного алгоритма (называемого по-английски **inorder tree walk**). Этот алгоритм печатает ключ корня поддерева после всех ключей его левого поддерева, но перед ключами правого поддерева. (Заметим в скобках, что порядок, при котором корень предшествует обоим поддеревьям, называется **preorder**; порядок, в котором корень следует за ними, называется **postorder**.)

Вызов INORDER-TREE-WALK(*root*[T]) печатает (в указанном порядке) все ключи, входящие в дерево T с корнем *root*[T].

```
INORDER-TREE-WALK($x$)
if $x \neq$ NIL
   then INORDER-TREE-WALK(left[$x$])
        напечатать key[$x$]
        INORDER-TREE-WALK(right[$x$])
```

К примеру, для обоих деревьев рис. 13.1 будет напечатано $2, 3, 5, 5, 7, 8$. Свойство упорядоченности гарантирует правильность алгоритма (индукция по высоте поддерева). Время работы на дереве с n вершинами есть $\Theta(n)$: на каждую вершину тратится ограниченное время (помимо рекурсивных вызовов) и каждая вершина обрабатывается один раз.

Упражнения

13.1-1 Нарисуйте двоичные деревья поиска высоты 2, 3, 4, 5 и 6 для одного и того же множества ключей $\{1, 4, 5, 10, 16, 17, 21\}$.

13.1-2 Кучи из раздела 7.1 также были двоичными деревьями, и требование упорядоченности там тоже было. В чём разница между тем требованием и теперешним? Как вы думаете, можно ли напечатать элементы двоичной кучи в неубывающем порядке за время $O(n)$? Объясните ваш ответ.

13.1-3 Напишите нерекурсивный алгоритм, печатающий ключи в двоичном дереве поиска в неубывающем порядке. (Указание: Простое решение использует в качестве дополнительной структуры стек; более изящное решение не требует стека, но предполагает, что можно проверять равенство указателей.)

13.1-4 Напишите рекурсивные алгоритмы для обхода деревьев в различных порядках (preorder, postorder). Как и раньше, время работы должно быть $O(n)$ (где n — число вершин).

13.1-5 Покажите, что любой алгоритм построения двоичного дерева поиска, содержащего заданные n элементов, требует (в худшем случае) времени $\Omega(n \lg n)$. Воспользуйтесь тем, что сортировка n чисел требует $\Omega(n \lg n)$ действий.

Рисунок 13.2 Поиск в двоичном дереве. Ища ключ 13, мы идём от корня по пути 15 → 6 → 7 → 13. Чтобы найти минимальный ключ 2, мы всё время идём налево; чтобы найти максимальный ключ 20 — направо. Для вершины с ключом 15 следующей будет вершина с ключом 17 (это минимальный ключ в правом поддереве вершины с ключом 15). У вершины с ключом 13 нет правого поддерева; поэтому, чтобы найти следующую за ней вершину, мы поднимаемся вверх, пока не пройдём по ребру, ведущему вправо-вверх; в данном случае следующая вершина имеет ключ 15.

13.2 Поиск в двоичном дереве

В этом разделе мы покажем, что двоичные деревья поиска позволяют выполнять операции Search, Minimum, Maximum, Successor и Predecessor за время $O(h)$, где h — высота дерева.

Поиск

Процедура поиска получает на вход искомый ключ k и указатель x на корень поддерева, в котором производится поиск. Она возвращает указатель на вершину с ключом k (если такая есть) или специальное значение NIL (если такой вершины нет).

```
Tree-Search($x, k$)
if $x =$ NIL или $k = key[x]$
    then return $x$
if $k < key[x]$
    then return Tree-Search($left[x], k$)
    else return Tree-Search($right[x], k$)
```

В процессе поиска мы двигаемся от корня, сравнивая ключ k с ключом, хранящимся в текущей вершине x. Если они равны, поиск завершается. Если $k < key[x]$, то поиск продолжается в левом поддереве x (ключ k может быть только там, согласно свойству упорядоченности). Если $k > key[x]$, то поиск продолжается в правом поддереве. Длина пути поиска не превосходит высоты дерева, поэтому время поиска есть $O(h)$ (где h — высота дерева).

Вот итеративная версия той же процедуры (которая, как правило, более эффективна):

```
ITERATIVE-TREE-SEARCH$(x, k)$
while $x \neq$ NIL и $k \neq key[x]$
    do if $k < key[x]$
        then $x \leftarrow left[x]$
        else $x \leftarrow right[x]$
return $x$
```

Минимум и максимум

Минимальный ключ в дереве поиска можно найти, пройдя по указателям *left* от корня (пока не упрёмся в NIL), см. рис. 13.2. Процедура возвращает указатель на минимальный элемент поддерева с корнем x.

```
TREE-MINIMUM$(x)$
while $left[x] \neq$ NIL
    do $x \leftarrow left[x]$
return $x$
```

Свойство упорядоченности гарантирует правильность процедуры TREE-MINIMUM. Если у вершины x нет левого ребёнка, то минимальный элемент поддерева с корнем x есть x, так как любой ключ в правом поддереве не меньше $key[x]$. Если же левое поддерево вершины x не пусто, то минимальный элемент поддерева с корнем x находится в этом левом поддереве (поскольку сам x и все элементы правого поддерева больше).

Алгоритм TREE-MAXIMUM симметричен:

```
TREE-MAXIMUM$(x)$
while $right[x] \neq$ NIL
    do $x \leftarrow right[x]$
return $x$
```

Оба алгоритма требуют времени $O(h)$, где h — высота дерева (поскольку двигаются по дереву только вниз).

Следующий и предыдущий элементы

Как найти в двоичном дереве элемент, следующий за данным? Свойство упорядоченности позволяет сделать это, двигаясь по дереву. Вот процедура, которая возвращает указатель на следующий за x элемент (если все ключи различны, он содержит следующий по величине ключ) или NIL, если элемент x — последний в дереве.

```
Tree-Successor($x$)
1  if $right[x] \neq$ NIL
2     then return Tree-Minimum($right[x]$)
3  $y \leftarrow p[x]$
4  while $y \neq$ NIL and $x = right[y]$
5      do $x \leftarrow y$
6         $y \leftarrow p[y]$
7  return $y$
```

Процедура Tree-Successor отдельно рассматривает два случая. Если правое поддерево вершины x непусто, то следующий за x элемент — минимальный элемент в этом поддереве и равен Tree-Minimum($right[x]$). Например, на рис. 13.2 за вершиной с ключом 15 следует вершина с ключом 17.

Пусть теперь правое поддерево вершины x пусто. Тогда мы идём от x вверх, пока не найдём вершину, являющуюся левым сыном своего родителя (строки 3–7). Этот родитель (если он есть) и будет искомым элементом. [Формально говоря, цикл в строках 4–6 сохраняет такое свойство: $y = p[x]$; искомый элемент непосредственно следует за элементами поддерева с корнем в x.]

Время работы процедуры Tree-Successor на дереве высоты h есть $O(h)$, так как мы двигаемся либо только вверх, либо только вниз.

Процедура Tree-Predecessor симметрична.

Таким образом, мы доказали следующую теорему.

Теорема 13.1. *Операции* Search, Minimum, Maximum, Successor *и* Predecessor *на дереве высоты h выполняются за время $O(h)$.*

Упражнения

13.2-1 Предположим, что в двоичном дереве поиска хранятся числа от 1 до 1000 и мы хотим найти число 363. Какие из следующих последовательностей *не* могут быть последовательностями просматриваемых при этом ключей?

а. $2, 252, 401, 398, 330, 344, 397, 363$;

б. $924, 220, 911, 244, 898, 258, 362, 363$;

в. $925, 202, 911, 240, 912, 245, 363$;

г. $2, 399, 387, 219, 266, 382, 381, 278, 363$;

д. $935, 278, 347, 621, 299, 392, 358, 363$.

13.2-2 Пусть поиск ключа в двоичном дереве завершается в листе. Рассмотрим три множества: A (элементы слева от пути поиска),

B (элементы на пути) и C (справа от пути). Профессор утверждает, что для любых трёх ключей $a \in A$, $b \in B$ и $c \in C$ верно $a \leqslant b \leqslant c$. Покажите, что он неправ, и приведите контрпример минимально возможного размера.

13.2-3 Докажите формально правильность процедуры Tree-Successor.

13.2-4 В разделе 13.1 был построен алгоритм, печатающий все ключи в неубывающем порядке. Теперь это можно сделать иначе: найти минимальный элемент, а потом $n-1$ раз искать следующий элемент. Докажите, что время работы такого алгоритма есть $O(n)$.

13.2-5 Докажите, что k последовательных вызовов Tree-Successor выполняются за $O(k+h)$ шагов (h — высота дерева) независимо от того, с какой вершины мы начинаем.

13.2-6 Пусть T — двоичное дерево поиска, все ключи в котором различны, x — его лист, а y — родитель x. Покажите, что $key[y]$ является соседним с $key[x]$ ключом (следующим или предыдущим в смысле порядка на ключах).

13.3 Добавление и удаление элемента

Эти операции меняют дерево, сохраняя свойство упорядоченности. Как мы увидим, добавление сравнительно просто; удаление чуть сложнее.

Добавление

Процедура Tree-Insert добавляет заданный элемент в подходящее место дерева T (сохраняя свойство упорядоченности). Параметром процедуры является указатель z на новую вершину, в которую помещены значения $key[z]$ (добавляемое значение ключа), $left[z] = \text{NIL}$ и $right[z] = \text{NIL}$. В ходе работы процедура меняет дерево T и (возможно) некоторые поля вершины z, после чего новая вершина с данным значением ключа оказывается вставленной в подходящее место дерева.

Рисунок 13.3 Добавление элемента с ключом 13. Светло-серые вершины находятся на пути от корня до позиции нового элемента. Пунктир связывает новый элемент со старыми.

```
Tree-Insert$(T, z)$
 1  $y \leftarrow$ NIL
 2  $x \leftarrow root[T]$
 3  while $x \neq$ NIL
 4       do $y \leftarrow x$
 5          if $key[z] < key[x]$
 6            then $x \leftarrow left[x]$
 7            else $x \leftarrow right[x]$
 8  $p[z] \leftarrow y$
 9  if $y =$ NIL
10     then $root[T] \leftarrow z$
11     else if $key[z] < key[y]$
12            then $left[y] \leftarrow z$
13            else $right[y] \leftarrow z$
```

На рисунке 13.3 показано, как работает процедура Tree-Insert. Подобно процедурам Tree-Search и Iterative-Tree-Search, она двигается вниз по дереву, начав с его корня. При этом в вершине y сохраняется указатель на родителя вершины x (цикл в строках 3–7). Сравнивая $key[z]$ с $key[x]$, процедура решает, куда идти — налево или направо. Процесс завершается, когда x становится равным NIL. Этот NIL стоит как раз там, куда надо поместить z, что и делается в строках 8–13.

Как и остальные операции, добавление требует времени $O(h)$ для дерева высоты h.

Удаление

Параметром процедуры удаления является указатель на удаляемую вершину. При удалении возможны три случая, показанные на рисунке 13.4. Если у z нет детей, для удаления z достаточно поместить NIL в соответствующее поле его родителя (вместо z). Если у z есть один ребёнок, можно *"вырезать"* z, соединив его родителя

Рисунок 13.4 Удаление вершины z из двоичного дерева поиска. (а) Если вершина z не имеет детей, её можно удалить без проблем. (б) Если вершина z имеет одного ребёнка, помещаем его на место вершины z. (в) Если у вершины z двое детей, мы сводим дело к предыдущему случаю, удаляя вместо неё вершину y с непосредственно следующим значением ключа (у этой вершины ребёнок один) и помещая ключ $key[y]$ (и связанные с ним дополнительные данные) на место вершины z.

напрямую с его ребёнком. Если же детей двое, требуются некоторые приготовления: мы находим следующий (в смысле порядка на ключах) за z элемент y; у него нет левого ребёнка (упр. 13.3-4). Теперь можно скопировать ключ и дополнительные данные из вершины y в вершину z, а саму вершину y удалить описанным выше способом.

Примерно так и действует процедура Tree-Delete (хотя рассматривает эти три случая в несколько другом порядке).

```
Tree-Delete($T$, $z$)
 1  if $left[z]$ = NIL или $right[z]$ = NIL
 2     then $y \leftarrow z$
 3     else $y \leftarrow$ Tree-Successor($z$)
 4  if $left[y] \neq$ NIL
 5     then $x \leftarrow left[y]$
 6     else $x \leftarrow right[y]$
 7  if $x \neq$ NIL
 8     then $p[x] \leftarrow p[y]$
 9  if $p[y]$ = NIL
10     then $root[T] \leftarrow x$
11     else if $y = left[p[y]]$
12             then $left[p[y]] \leftarrow x$
13             else $right[p[y]] \leftarrow x$
14  if $y \neq z$
15     then $key[z] \leftarrow key[y]$
16            ▷ копируем дополнительные данные, связанные с $y$
17  return $y$
```

В строках 1–3 определяется вершина y, которую мы потом вырежем из дерева. Это либо сама вершина z (если у z не более одного ребёнка), либо следующий за z элемент (если у z двое детей). Затем в строках 4–6 переменная x становится указателем на существующего ребёнка вершины y, или равной NIL, если у y нет детей. Вершина y вырезается из дерева в строках 7–13 (меняются указатели в вершинах $p[y]$ и x). При этом отдельно рассматриваются граничные случаи, когда $x =$ NIL и когда y является корнем дерева. Наконец, в строках 14–16, если вырезанная вершина y отлична от z, ключ (и дополнительные данные) вершины y перемещаются в z (ведь нам надо было удалить z, а не y). Наконец, процедура возвращает указатель y (это позволит вызывающей процедуре впоследствии освободить память, занятую вершиной y). Время выполнения есть $O(h)$ на дереве высоты h.

Итак, мы доказали следующую теорему.

Теорема 13.2. *Операции* Insert *и* Delete *могут быть выполнены за время $O(h)$, где h — высота дерева.*

Упражнения

13.3-1 Напишите рекурсивный вариант процедуры Tree-Insert.

13.3-2 Начиная с пустого дерева, будем добавлять элементы с различными ключами один за другим. Если после этого мы проводим поиск элемента с ключом x, то число сравнений на единицу больше числа сравнений, выполненных при добавлении этого элемента. Почему?

13.3-3 Набор из n чисел можно отсортировать, сначала добавив их один за другим в двоичное дерево поиска (с помощью процедуры Tree-Insert), а потом обойти дерево с помощью процедуры Inorder-Tree-Walk. Найдите время работы такого алгоритма в худшем и в лучшем случае.

13.3-4 Покажите, что, если вершина двоичного дерева поиска имеет двоих детей, то следующая за ней вершина не имеет левого ребёнка, а предшествующая ей вершина — правого.

13.3-5 Предположим, что указатель на вершину y хранится в какой-то внешней структуре данных и что предшествующая y вершина дерева удаляется с помощью процедуры Tree-Delete. Какие при этом могут возникнуть проблемы? Как можно изменить Tree-Delete, чтобы этих проблем избежать?

13.3-6 Коммутируют ли операции удаления двух вершин? Другими словами, получим ли мы одинаковые деревья, если в одном случае удалим сначала x, а потом y, а в другом — наоборот? Объясните свой ответ.

13.3-7 Если у z двое детей, мы можем использовать в Tree-Delete не следующий за z элемент, а предыдущий. Можно надеяться, что справедливый подход, который в половине случаев выбирает предыдущий, а в половине — следующий элемент, будет приводить к лучше сбалансированному дереву. Как изменить текст процедуры, чтобы реализовать такой подход?

⋆ 13.4 Случайные двоичные деревья поиска

Как мы видели, основные операции с двоичными деревьями поиска требуют времени $O(h)$, где h — высота дерева. Поэтому важно понять, какова высота *"типичного"* дерева. Для этого необходимо принять какие-то статистические предположения о распределении ключей и последовательности выполняемых операций.

К сожалению, в общем случае ситуация трудна для анализа, и мы будем рассматривать лишь деревья, полученные добавлением вершин (без удалений). Определим **случайное двоичное дерево** (randomly built search tree) из n различных ключей как дерево, получающееся из пустого дерева добавлением этих ключей в случайном порядке (все $n!$ перестановок считаем равновероятными). (Как видно из упр. 13.4-2, это не означает, что все двоичные деревья равновероятны, поскольку разные порядки добавления могут приводить к одному и тому же дереву.) В этом разделе мы докажем, что математическое ожидание высоты случайного дерева из n ключей есть $O(\lg n)$.

Посмотрим, как связана структура дерева с порядком добавления ключей.

Лемма 13.3. *Пусть T — дерево, получающееся после добавления n различных ключей $k_1, k_2, \ldots, k_n$ (в указанном порядке) к изначально пустому дереву. Тогда k_i является предком k_j в T тогда и только тогда, когда $i < j$, и при этом*

$$k_i = \min\{k_l : 1 \leqslant l \leqslant i \text{ и } k_l > k_j\}$$

(ключ k_i больше k_j и их не разделяет ни один ключ среди $k_1, \ldots, k_i$) или

$$k_i = \max\{k_l : 1 \leqslant l \leqslant i \text{ и } k_l < k_j\}$$

(ключ k_i меньше k_j и их не разделяет ни один ключ среди $k_1, \ldots, k_i$)

Доказательство. $\Rightarrow$: Предположим, что k_i является предком k_j. Очевидно, $i < j$ (потомок появляется в дереве позже предка). Рассмотрим дерево T_i, которое получается после добавления ключей $k_1, k_2, \ldots, k_i$. Путь в T_i от корня до k_i тот же, что и путь в T от корня до k_i. Таким образом, если бы ключ k_j был добавлен в T_i, он стал бы правым или левым ребёнком k_i. Следовательно (см. упр. 13.2-6), k_i является либо наименьшим среди тех ключей из $k_1, k_2, \ldots, k_i$, которые больше k_j, либо наибольшим среди ключей из того же набора, меньших k_j.

$\Leftarrow$: Предположим, что k_i является наименьшим среди тех ключей $k_1, k_2, \ldots, k_i$, которые больше k_j. (Другой случай симметричен.) Что будет происходить при помещении ключа k_j в дерево? Сравнение k_j с ключами на пути от корня к k_i даст те же результаты, что и для k_i. Следовательно, мы пройдём путь от корня до k_i, так что k_j станет потомком k_i.

Лемма доказана. □

Теперь можно понять, как зависит глубина каждого ключа от перестановки на входе.

Следствие 13.4. Пусть T — дерево, полученное из пустого добавлением n различных ключей $k_1, k_2, \ldots, k_n$ (в указанном порядке). Для каждого ключа k_j (при всех $1 \leqslant j \leqslant n$) рассмотрим множества

$$G_j = \{k_i : 1 \leqslant i < j \text{ и } k_l > k_i > k_j \text{ при всех } l < i, \text{ для которых } k_l > k_j\}$$

и

$$L_j = \{k_i : 1 \leqslant i < j \text{ и } k_l < k_i < k_j \text{ при всех } l < i, \text{ для которых } k_l < k_j\}.$$

Тогда ключи на пути из корня в k_j — в точности $G_j \cup L_j$, а глубина k_j в дереве T равна

$$d(k_j, T) = |G_j| + |L_j|.$$

На рисунке 13.5 изображены множества G_j и L_j. Их построение можно объяснить так. Считая для наглядности ключи числами, будем отмечать их на числовой оси: сначала k_1, потом k_2 и так далее вплоть до k_j. В каждый момент на оси отмечено несколько точек $k_1, \ldots, k_t$ (где $1 \leqslant t \leqslant j-1$). Посмотрим, какая из этих точек будет ближайшей справа к будущему положению ключа k_j. Множество всех таких ближайших точек (для всех моментов времени $t = 1, 2, \ldots, j-1$) и есть G_j. Ближайшие слева точки образуют множество L_j.

Наша цель — оценить сверху количество элементов в G_j и L_j, поскольку сумма этих количеств равна глубине ключа k_j. Фиксируем некоторое j. Число элементов в G_j будет случайной величиной, зависящей от порядка ключей на входе (имеет значение лишь порядок на ключах $k_1, \ldots, k_j$). Мы хотим оценить это число сверху (доказать, что вероятность события *"это число велико"* мала).

Следующий факт из теории вероятностей играет центральную роль при этой оценке.

Лемма 13.5. *Пусть $k_1, k_2, \ldots, k_n$ есть случайная перестановка n различных чисел. Для каждого i от 1 до n рассмотрим минимальный элемент в множестве $\{k_1, k_2, \ldots, k_i\}$. Множество всех таких элементов назовём S:*

$$S = \{k_i : 1 \leqslant i \leqslant n \text{ и } k_l > k_i \text{ для всех } l < i\}. \tag{13.1}$$

Тогда $\mathsf{P}\{|S| \geqslant (\beta+1)H_n\} \leqslant 1/n^2$, где H_n — n-я частичная сумма гармонического ряда, а $\beta \approx 4{,}32$ — корень уравнения $(\ln\beta - 1)\beta = 2$.

Доказательство. Будем следить за тем, как меняется множество $\{k_1, \ldots, k_i\}$ с ростом i. На i-м шаге к нему добавляется элемент k_i, причём он с равными вероятностями может оказаться первым, вторым,..., i-м по величине (каждой из этих возможностей соответствует равная доля входных перестановок). Таким образом, вероятность увеличения множества S на i-м шаге равна $1/i$ при любом порядке среди ключей $k_1, \ldots, k_{i-1}$, так что для разных i эти события независимы.

Мы приходим к ситуации, описанной в теореме 6.6: имеется последовательность независимых испытаний, вероятность успеха в i-м испытании равна $1/i$. Нам надо оценить число успехов.

Математическое ожидание этого числа равно $1 + 1/2 + \ldots + 1/i = H_i = \ln i + O(1)$, см. формулу (3.5) и задачу 6-2. Нам надо оценить вероятность того, что число успехов больше своего математического ожидания в $\beta + 1$ раз.

Это делается с помощью теоремы 6.6. Напомним, что математическое ожидание $|S|$ есть $\mu = H_n \geqslant \ln n$, и по теореме 6.6 мы имеем

$$\begin{aligned}
\mathsf{P}\{|S| \geqslant (\beta+1)H_n\} &= P\{|S| - \mu \geqslant \beta H_n\} \leqslant \\
&\leqslant \left(\frac{eH_n}{\beta H_n}\right)^{\beta H_n} = \\
&= e^{(1-\ln\beta)\beta H_n} \leqslant \\
&\leqslant e^{-(\ln\beta - 1)\beta \ln n} = \\
&= n^{-(\ln\beta-1)\beta} = \\
&= 1/n^2
\end{aligned}$$

согласно определению числа β. □

После такой подготовки вернёмся к деревьям поиска.

Теорема 13.6. *Средняя высота случайного двоичного дерева поиска, построенного по n различным ключам, есть $O(\lg n)$.*

Доказательство. Пусть $k_1, k_2, \ldots, k_n$ — случайная перестановка данных n ключей, T — дерево, полученное последовательным добавлением этих ключей к пустому. Для фиксированного номера j и для произвольного числа t рассмотрим вероятность того, что глубина $d(k_j, T)$ ключа k_j не меньше t. Согласно следствию 13.4, в этом случае хотя бы одно из множеств G_j и L_j должно иметь размер не менее $t/2$. Таким образом,

$$\mathsf{P}\{d(k_j, T) \geqslant t\} \leqslant \mathsf{P}\{|G_j| \geqslant t/2\} + \mathsf{P}\{|L_j| \geqslant t/2\}. \qquad (13.2)$$

Вначале рассмотрим $\mathsf{P}\{|G_j| \geqslant t/2\}$. Оценим условную вероятность этого события при фиксированном множестве $U = \{t : 1 \leqslant t \leqslant j - 1 \text{ и } k_t > k_j\}$ (то есть когда известно, какие из элементов $k_1, \ldots, k_{j-1}$ больше k_j). Мы находимся в ситуации леммы 13.5 (все перестановки элементов с индексами из U равновероятны), и поэтому условная вероятность события $|G_j| \geqslant t/2$ при данном U равна вероятности того, что в случайной перестановке из $u = |U|$ элементов есть по крайней мере $t/2$ элементов, меньших всех предыдущих. С ростом u эта вероятность только растёт, так что все условные вероятности не превосходят $\mathsf{P}\{|S| \geqslant t/2\}$, где S определено как в лемме 13.5. Поэтому и полная вероятность события $\{|G_j| \geqslant t/2\}$ не превосходит $\mathsf{P}\{|S| \geqslant t/2\}$.

Аналогичным образом

$$\mathsf{P}\{|L_j| \geqslant t/2\} \leqslant \mathsf{P}\{|S| \geqslant t/2\}$$

и, согласно неравенству (13.2),

$$\mathsf{P}\{d(k_j, T) \geqslant t\} \leqslant 2\mathsf{P}\{|S| \geqslant t/2\}.$$

Взяв теперь $t = 2(\beta+1)H_n$, где $H_n = 1+1/2+\ldots+1/n$ (а $\beta \approx 4{,}32$ — корень уравнения $(\ln\beta - 1)\beta = 2$) и применив лемму 13.5, мы заключаем, что

$$\mathsf{P}\{d(k_j,T) \geqslant 2(\beta+1)H_n\} \leqslant 2\mathsf{P}\{|S| \geqslant (\beta+1)H_n\} \leqslant 2/n^2.$$

Всего вершин в дереве не более n, поэтому вероятность того, что какая-то вершина будет иметь глубину $2(\beta+1)H_n$ или больше, не более чем в n раз превосходит такую же вероятность для одной вершины, и потому не превосходит $2/n$. Итак, с вероятностью по меньшей мере $1-2/n$ высота случайного дерева не превосходит $2(\beta+1)H_n$, и в любом случае она не больше n. Таким образом, математическое ожидание не превосходит $(2(\beta+1)H_n)(1-2/n)+n(2/n) = O(\lg n)$. □

Упражнения

13.4-1 Приведите пример дерева поиска, в котором средняя (по всем вершинам) глубина вершины есть $\Theta(\lg n)$, но высота дерева есть $\omega(\lg n)$. Насколько велика может быть высота дерева, если средняя глубина вершины есть $\Theta(\lg n)$?

13.4-2 Покажите, что при нашем понимании случайного двоичного дерева поиска не все упорядоченные деревья с данными n ключами равновероятны. (Указание: Рассмотрите случай $n = 3$.)

13.4-3* Для данной константы $r \geqslant 1$ укажите константу t, для которой вероятность события *"высота случайного двоичного дерева поиска не меньше tH_n"* меньше $1/n^r$.

13.4-4* Рассмотрим алгоритм RANDOMIZED-QUICKSORT, применённый к последовательности из n чисел. Докажите, что для любой константы $k > 0$ существует такая константа c, что с вероятностью не менее $1 - c/n^k$ алгоритм завершает работу за время $cn\lg n$.

Задачи

13-1 Двоичные деревья поиска и равные ключи

Равные ключи — источник проблем при работе с деревьями поиска.

а Какова асимптотика времени работы процедуры TREE-INSERT при добавлении n одинаковых ключей в изначально пустое дерево?

Причина тут в том, что при выборе в строках 5–7 и 11–13 мы в случае равенства всегда двигаемся направо по дереву. Будем рассматривать случай равенства отдельно. Оцените асимптотику времени добавления n равных ключей в пустое дерево при использовании трёх различных подходов:

б Храним в вершине x флаг $b[x]$, и выбираем левого или правого ребёнка в зависимости от значения $b[x]$. При этом флаг меняется при каждом посещении вершины, так что направления чередуются.

в Храним элементы с равными ключами в одной вершине (с помощью списка) и добавляем элемент с уже встречавшимся ключом в этот список.

г Направление движения выбираем случайно. (Каково будет время в худшем случае? Что вы можете сказать о математическом ожидании?)

13-2 Цифровые деревья

Рассмотрим две строки $a = a_0a_1 \ldots a_p$ и $b = b_0b_1 \ldots b_q$, составленные из символов некоторого (упорядоченного) алфавита. Говорят, что строка a **лексикографически меньше** строки b (a is lexicographically less than b), если выполняется одно из двух условий:

1. Существует число j из $0..\min(p,q)$, при котором $a_i = b_i$ для всех $i = 0, 1, \ldots, j-1$ и $a_j < b_j$.
2. $p < q$ и $a_i = b_i$ для всех $i = 0, 1, \ldots, p$.

Например, $10100 < 10110$ согласно правилу 1 (при $j = 3$), а $10100 < 101000$ согласно правилу 2. Такой порядок применяется в словарях.

Строение **цифрового дерева** (radix tree) видно из примера на рис. 13.6, где показано дерево, хранящее битовые строки 1011, 10, 011, 100 и 0. При поиске строки $a = a_0a_1 \ldots a_p$ мы на i-м шаге идём налево при $a_i = 0$ и направо при $a_i = 1$. Пусть элементами множества S являются попарно различные битовые строки суммарной длины n. Покажите, как с помощью цифрового дерева отсортировать S в лексикографическом порядке за $\Theta(n)$ действий. (Например, для множества рис. 13.6 результатом сортировки будет последовательность $0, 011, 10, 100, 1011$.)

13-3 Средняя глубина вершины в случайном двоичном дереве

В этой задаче мы докажем, что математическое ожидание средней глубины вершины в случайном двоичном дереве с n вершинами есть $O(\lg n)$. Хотя этот результат слабее, чем результат теоремы 13.6, доказательство устанавливает интересные аналогии между двоичными деревьями поиска и процедурой RANDOMIZED-QUICKSORT из раздела 8.3.

Рассмотрим (ср. главу 5, упр. 5.5-6) сумму глубин $d(x,T)$ всех вершин x дерева T, которую мы будем обозначать $P(T)$.

а Средняя глубина вершины в дереве T есть

$$\frac{1}{n}\sum_{x\in T} d(x,T) = \frac{1}{n}P(T).$$

Таким образом, надо показать, что математическое ожидание $P(T)$ есть $O(n\lg n)$.

б Обозначим через T_L и T_R левое и правое поддеревья дерева T. Убедитесь, что если T содержит n вершин, то

$$P(T) = P(T_L) + P(T_R) + n - 1.$$

в Обозначим через $P(n)$ математическое ожидание внутренней суммы длин для случайного двоичного дерева поиска с n вершинами. Покажите, что

$$P(n) = \frac{1}{n}\sum_{i=0}^{n-1}(P(i) + P(n-i-1) + n - 1).$$

г Покажите, что

$$P(n) = \frac{2}{n}\sum_{k=1}^{n-1} P(k) + \Theta(n).$$

д Вспоминая рассуждение из раздела 8.4.2 (оценка математического ожидания времени быстрой сортировки), покажите, что $P(n) = O(n\lg n)$.

В каждом рекурсивном вызове быстрой сортировки мы случайным образом выбираем граничный элемент. Подобно этому, каждая вершина дерева поиска является границей между левым и правым своим поддеревом.

е Опишите реализацию алгоритма быстрой сортировки, при котором в процессе сортировки элементов $k_1,\ldots,k_n$ выполняются в точности те же сравнения, что и при добавлении их в (изначально пустое) дерево. (Порядок сравнений может быть другим.)

13-4 Количество разных двоичных деревьев

Обозначим через b_n количество различных двоичных деревьев с n вершинами. В этой задаче требуется вывести формулу для b_n и оценить скорость роста числа b_n.

а Покажите, что $b_0 = 1$ и что

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k}$$

при $n \geqslant 1$.

б Пусть $B(x)$ — производящая функция

$$B(x) = \sum_{n=0}^{\infty} b_n x^n$$

(определение производящих функций дано в задаче 4-6). Покажите, что $B(x) = xB(x)^2 + 1$ и, таким образом,

$$B(x) = \frac{1}{2x}(1 - \sqrt{1-4x}).$$

Ряд Тейлора (Taylor expansion) функции $f(x)$ в точке $x = a$ определяется формулой

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k$$

где $f^{(k)}(a)$ — k-я производная f в точке a.

в Покажите, что

$$b_n = \frac{1}{n+1} C_{2n}^n$$

разложив в ряд Тейлора функцию $\sqrt{1-4x}$ в точке 0. (Ряд для $\sqrt{1+h} = (1+h)^{1/2}$ можно получить также как обобщение бинома Ньютона, если для нецелых n и целых неотрицательных k положить $C_n^k = n(n-1)\ldots(n-k+1)/k!$.)

Число b_n называется n-м **числом Каталана** (Catalan number).

г Покажите, что

$$b_n = \frac{4^n}{\sqrt{\pi}n^{3/2}}(1 + O(1/n)).$$

Замечания

Подробное обсуждение двоичных деревьев поиска и многих аналогичных структур данных можно найти у Кнута [123]. Видимо, двоичные деревья поиска были независимо придуманы многими людьми незадолго до 1960 года.

ключи	21	9	4	25	7	12	3	10	19	29	17	6	26	18
G'_j	21			25					19	29				
G_j	21								19					
L'_j		9	4		7	12	3	10						
L_j		9				12								

(б)

Рисунок 13.5 Множества G_j и L_j, составляющие множество ключей на пути к ключу $k_j = 17$. (а) Черные вершины содержат ключи из G_j, белые из L_j, остальные вершины — серые. Выделен путь к ключу k_j. Ключи левее пунктирной линии меньше k_j, ключи правее — больше. (б) Множество $G'_j = \{21, 25, 19, 29\}$ состоит из ключей, добавленных раньше ключа 17 и больших 17. Множество $G_j = \{21, 19\}$ содержит ключи, бывшие ближайшими справа к ключу 17, то есть бывшие минимальными в уже появившейся части G'_j. Ключ 21 был добавлен первым к G'_j и попал в G_j; ключ 25 не попал (он больше текущего минимума, равного 21). Ключ 19 попадает в G_j, потому что он меньше 21, а 29 — нет, так как $29 > 19$. Множества L'_j и L_j строятся аналогичным образом.

Рисунок 13.6 Цифровое дерево хранит строки 1011, 10, 011, 100 и 0. Каждой вершине соответствует ключ — строка, которую можно прочесть, идя из корня в эту вершину, поэтому эти ключи не надо хранить специально (на рисунке они показаны для наглядности). Тёмные вершины не соответствуют ключам, а являются промежуточными для других вершин.

14 Красно-чёрные деревья

В главе 13 мы показали, что основные операции с двоичным деревом поиска высоты h могут быть выполнены за $O(h)$ действий. Деревья эффективны, если их высота мала — но малая высота не гарантируется, и в худшем случае деревья не более эффективны, чем списки. Красно-чёрные деревья — один из типов *"сбалансированных"* деревьев поиска, в которых предусмотрены операции балансировки, гарантирующие, что высота дерева не превзойдёт $O(\lg n)$.

14.1 Свойства красно-чёрных деревьев

Красно-чёрное дерево (red-black tree) — это двоичное дерево поиска, вершины которого разделены на красные (red) и чёрные (black). Таким образом, каждая вершина хранит один дополнительный бит — её цвет.

При этом должны выполняться определённые требования, которые гарантируют, что глубины любых двух листьев отличаются не более чем в два раза, поэтому дерево можно назвать **сбалансированным** (balanced).

Каждая вершина красно-чёрного дерева имеет поля *color* (цвет), *key* (ключ), *left* (левый ребёнок), *right* (правый ребёнок) и p (родитель). Если у вершины отсутствует ребёнок или родитель, соответствующее поле содержит NIL. Для удобства мы будем считать, что значения NIL, хранящиеся в полях *left* и *right*, являются ссылками на дополнительные (фиктивные) листья дерева. В таком пополненном дереве каждая старая вершина (содержащая ключ) имеет двух детей.

Двоичное дерево поиска называется красно-чёрным деревом, если оно обладает следующими свойствами (будем называть их **RB-свойствами**, по-английски red-black properties):

1. Каждая вершина — либо красная, либо чёрная.
2. Каждый лист (NIL) — чёрный.

Рисунок 14.1 Красно-чёрное дерево. Чёрные вершины показаны как тёмные, красные — как серые. Каждая вершина либо красная, либо черная. Все NIL-листья чёрные. Дети красной вершины — чёрные. Для каждой вершины все пути от неё вниз к листьям содержит одинаковое количество чёрных вершин. Около каждой вершины (кроме листьев) записана её чёрная высота. Чёрная высота листьев равна 0.

3. Если вершина красная, оба её ребёнка чёрные.
4. Все пути, идущие вниз от корня к листьям, содержат одинаковое количество чёрных вершин.

Пример красно-чёрного дерева показан на рисунке 14.1.

Рассмотрим произвольную вершину x красно-чёрного дерева и пути, ведущие вниз от неё к листьям. Все они содержат одно и то же число чёрных вершин (добавим к ним путь из корня в x и применим свойство 4). Число чёрных вершин в любом из них (саму вершину x мы не считаем) будем называть **чёрной высотой** (black-height) вершины x и обозначать $\mathrm{bh}(x)$. Чёрной высотой дерева будем считать чёрную высоту его корня.

Следующая лемма показывает, что красно-чёрные деревья хороши как деревья поиска.

Лемма 14.1. *Красно-чёрное дерево с n внутренними вершинами (т. е. не считая* NIL*-листьев) имеет высоту не больше* $2\lg(n+1)$.

Доказательство. Сначала покажем, что поддерево с корнем в x содержит по меньшей мере $2^{\mathrm{bh}(x)} - 1$ внутренних вершин. Доказательство проведём индукцией от листьев к корню. Для листьев чёрная высота равна 0, и поддерево в самом деле содержит не менее $2^{\mathrm{bh}(x)} - 1 = 2^0 - 1 = 0$ внутренних вершин. Пусть теперь вершина x не является листом и имеет чёрную высоту k. Тогда оба её ребёнка имеют чёрную высоту не меньше $k - 1$ (красный ребёнок будет иметь высоту k, чёрный — $k - 1$). По предположению индукции левое и правое поддеревья вершины x содержат не менее $2^{k-1} - 1$ вершин, и потому поддерево с корнем в x содержит по меньшей мере $2^{k-1} - 1 + 2^{k-1} - 1 + 1 = 2^k - 1$ внутренних вершин.

Чтобы завершить доказательство леммы, обозначим высоту дерева через h. Согласно свойству 3, по меньшей мере половину всех вершин на пути от корня к листу, не считая корень, составляют чёрные вершины. Следовательно, чёрная высота дерева не меньше $h/2$. Тогда

$$n \geqslant 2^{h/2} - 1.$$

Перенося 1 налево и перейдя к логарифмам, получаем $\lg(n+1) \geqslant h/2$, или $h \leqslant 2\lg(n+1)$. Лемма доказана. □

Тем самым для красно-чёрных деревьев операции SEARCH, MINIMUM, MAXIMUM, SUCCESSOR и PREDECESSOR выполняются за время $O(\lg n)$, так как время их выполнения есть $O(h)$ для дерева высоты h, а красно-чёрное дерево с n вершинами имеет высоту $O(\lg n)$.

Сложнее обстоит дело с процедурами TREE-INSERT и TREE-DELETE из главы 13: проблема в том, что они могут испортить структуру красно-чёрного дерева, нарушив RB-свойства. Поэтому эти процедуры придётся модифицировать. Мы увидим в разделах 14.3 и 14.4, как можно реализовать добавление и удаление элементов за время $O(\lg n)$ с сохранением RB-свойств.

Упражнения

14.1-1 Нарисуйте полное двоичное дерево поиска высоты 3 с ключами $\{1, 2, \ldots, 15\}$. Добавьте NIL-листья и покрасьте вершины тремя способами так, чтобы получившиеся красно-чёрные деревья имели чёрную высоту 2, 3 и 4.

14.1-2 Предположим, что корень красно-чёрного дерева красный. Если мы покрасим его в чёрный цвет, останется ли дерево красно-чёрным?

14.1-3 Покажите, что самый длинный путь вниз от вершины x к листу не более чем вдвое длиннее самого короткого такого пути.

14.1-4 Какое наибольшее и наименьшее количество внутренних вершин может быть в красно-чёрном дереве чёрной высоты k?

14.1-5 Опишите красно-чёрное дерево, содержащее n ключей, с наибольшим возможным отношением числа красных внутренних вершин к числу чёрных внутренних вершин. Чему равно это отношение? Для какого дерева это отношение будет наименьшим, и чему равно это отношение?

Рисунок 14.2 Операции вращения на двоичном дереве поиска. Операция Right-Rotate преобразует левое дерево в правое, меняя несколько указателей. Правое дерево можно перевести в левое обратной операцией Left-Rotate. Вершины x и y могут находиться в любом месте дерева. Буквы α, β и γ обозначают поддеревья. В обоих деревьях выполнено одно и то же свойство упорядоченности: ключи из α предшествуют $key[x]$, который предшествует ключам из β, которые предшествуют $key[y]$, который предшествует ключам из γ.

14.2 Вращения

Операции Tree-Insert и Tree-Delete выполняются на красно-чёрном дереве за время $O(\lg n)$, но они изменяют дерево, и результат может не обладать RB-свойствами, описанными в разделе 14.1. Чтобы восстановить эти свойства, надо перекрасить некоторые вершины и изменить структуру дерева.

Мы будем менять структуру с помощью **вращений** (rotations). Вращение представляет собой локальную операцию (меняется несколько указателей) и сохраняет свойство упорядоченности. На рисунке 14.2 показаны два взаимно обратных вращения: левое и правое. Левое вращение возможно в любой вершине x, правый ребёнок которой (назовём его y) не является листом (NIL). После вращения y оказывается корнем поддерева, x — левым ребёнком y, а бывший левый ребёнок y — правым ребёнком x.

В процедуре Left-Rotate предполагается, что $right[x] \neq$ NIL.

```
Left-Rotate($T, x$)
$y \leftarrow right[x]$              ▷ Находим $y$.
$right[x] \leftarrow left[y]$        ▷ Левое поддерево $y$ становится
                                         правым поддеревом $x$.
if $left[y] \neq$ NIL
   then $p[left[y]] \leftarrow x$
$p[y] \leftarrow p[x]$               ▷ Делаем родителя $x$ родителем $y$.
if $p[x] =$ NIL
   then $root[T] \leftarrow y$
   else if $x = left[p[x]]$
           then $left[p[x]] \leftarrow y$
           else $right[p[x]] \leftarrow y$
$left[y] \leftarrow x$               ▷ Делаем $x$ левым ребёнком $y$.
$p[x] \leftarrow y$
```

На рисунке 14.3 показано действие процедуры Left-Rotate. Про-

Рисунок 14.3 Пример действия процедуры Left-Rotate. Дополнительные nil-листья не показаны. Порядок ключей в начальном и конечном деревьях один и тот же.

цедура Right-Rotate аналогична. Обе они работают за время $O(1)$ и меняют только указатели. Остальные поля вершин остаются неизменными.

Упражнения

14.2-1 Нарисуйте дерево, которое получится, если с помощью процедуры Tree-Insert добавить ключ 36 к дереву рис. 14.1. Если сделать добавленную вершину красной, будет ли полученное дерево обладать RB-свойствами? А если сделать её чёрной?

14.2-2 Напишите процедуру Right-Rotate.

14.2-3 Убедитесь, что вращения сохраняют свойство упорядоченности.

14.2-4 Пусть a, b и c — произвольные вершины в поддеревьях α, β и γ на рис. 14.2 (справа). Как изменится глубина a, b и c при выполнении левого вращения?

14.2-5 Покажите, что произвольное двоичное дерево поиска с n вершинами может быть преобразовано в любое другое дерево с тем же числом вершин (и теми же ключами) с помощью $O(n)$ вращений. (Указание: Сначала покажите, что $n-1$ правых вращений достаточно, чтобы преобразовать любое дерево в идущую вправо цепочку.)

14.3 Добавление вершины

Добавление вершины в красно-чёрное дерево проводится за время $O(\lg n)$. Сначала мы применяем процедуру Tree-Insert, как делалось для двоичных деревьев поиска, и красим новую вершину в красный цвет. После этого надо восстановить RB-свойства, для чего приходится перекрасить некоторые вершины и произвести вращения. При этом возможны различные ситуации, с которыми надо аккуратно разобраться.

```
RB-Insert$(T, x)$
 1  Tree-Insert$(T, x)$
 2  $color[x] \leftarrow$ RED
 3  while $x \neq root[T]$ и $color[p[x]] =$ RED
 4      do if $p[x] = left[p[p[x]]]$
 5            then $y \leftarrow right[p[p[x]]]$
 6                 if $color[y] =$ RED
 7                   then $color[p[x]] \leftarrow$ BLACK           ▷ Случай 1
 8                        $color[y] \leftarrow$ BLACK              ▷ Случай 1
 9                        $color[p[p[x]]] \leftarrow$ RED          ▷ Случай 1
10                        $x \leftarrow p[p[x]]$                   ▷ Случай 1
11                   else if $x = right[p[x]]$
12                          then $x \leftarrow p[x]$               ▷ Случай 2
13                               Left-Rotate$(T, x)$               ▷ Случай 2
14                        $color[p[x]] \leftarrow$ BLACK           ▷ Случай 3
15                        $color[p[p[x]]] \leftarrow$ RED          ▷ Случай 3
16                        Right-Rotate$(T, p[p[x]])$               ▷ Случай 3
17            else (аналогичный текст с
                        заменой $left \leftrightarrow right$)
18  $color[root[T]] \leftarrow$ BLACK
```

На рисунке 14.4 показан пример применения процедуры RB-Insert.

Процедура RB-Insert проще, чем кажется на первый взгляд. После выполнения строк 1–2 выполняются все RB-свойства, кроме одного: красная вершина x может иметь красного родителя. (см. рис. 14.4a). В остальном всё в порядке — другие свойства *"не замечают"* добавления красной вершины (отметим, что новая красная вершина имеет двух чёрных NIL-детей).

Такая ситуация (выполнены все RB-свойства, за исключением того, что красная вершина x может иметь красного родителя) будет сохраняться после любого числа итераций цикла. На каждом шаге вершина x поднимается вверх по дереву (если только не удалось устранить нарушения полностью; в этом случае мы выходим из цикла).

Рисунок 14.4 Работа процедуры RB-Insert. (а) Добавлена вершина x; при этом нарушилось свойство 3: x и его родитель красные. Вершина y (которую можно назвать *"дядей"* вершины x) красная, поэтому имеет место случай 1. После перекрашивания вершин получается дерево (б). Новая вершина x и её родитель красные, но дядя y чёрный. Так как x — правый ребёнок, имеет место случай 2. Производится левое вращение, которое дает дерево (в). Теперь уже x является левым ребёнком, и это — случай 3. После правого вращения получаем корректное красно-чёрное дерево (г).

Рисунок 14.5 Случай 1. Нарушено свойство 3: x и его родитель — красные. Наши действия не зависят от того, является ли x правым (а) или левым (б) сыном. Все поддеревья α, β, γ, δ и ε имеют чёрный корень и одинаковую чёрную высоту. Процедура меняет цвет вершины $p[p[x]]$ и её детей. Цикл продолжается после присваивания $x \leftarrow p[p[x]]$; свойство 3 может быть нарушено только между красной вершиной $p[p[x]]$ и её родителем (если он тоже красный).

Внутри цикла рассматриваются шесть случаев, но три из них симметричны трём другим, различия лишь в том, является ли родитель вершины x левым или правым ребёнком своего родителя. Эти случаи разделяются в строке 4. Мы выписали фрагмент процедуры для случая, когда $p[x]$ — левый ребёнок своего родителя. (Симметричные случаи относятся к строке 17.)

Мы будем предполагать, что во всех рассматриваемых нами красно-чёрных деревьях корень чёрный (и поддерживать это свойство — для этого используется строка 18). Поэтому в строке 4 красная вершина $p[x]$ не может быть корнем ($p[p[x]]$ существует).

Операции внутри цикла начинаются с нахождения вершины y, которая является *"дядей"* вершины x (имеет того же родителя, что и вершина $p[x]$). Если вершина y красная, имеет место случай 1, если чёрная — то один из случаев 2 и 3. Во всех случаях вершина $p[p[x]]$ чёрная, так как пара x–$p[x]$ была единственным нарушением RB-свойств.

Случай 1 (строки 7–10) показан на рис. 14.5. Эта часть текста исполняется, если и $p[x]$, и y красные. При этом вершина $p[p[x]]$ — чёрная. Перекрасим $p[x]$ и y в чёрный цвет, а $p[p[x]]$ — в красный. При этом число чёрных вершин на любом пути из корня к листьям останется прежним. Нарушение RB-свойства возможно в единственном месте нового дерева: у $p[p[x]]$ может быть красный родитель. Поэтому надо продолжить выполнение цикла, присвоив x значение $p[p[x]]$.

В случаях 2 и 3 вершина y чёрная. Эти два случая различаются тем, каким ребёнком x приходится своему родителю — левым или

Рисунок 14.6 Случаи 2 и 3 в процедуре RB-Insert. Как и для случая 1, нарушено свойство 3 красно-чёрных деревьев, так как вершина x и её родитель $p[x]$ — красные. Корни деревьев α, β, γ и δ — чёрные; эти деревья имеют одинаковую чёрную высоту. Оба вращения, показанные на рисунке, не меняют число чёрных вершин на пути от корня к листьям. После этого мы выходим из цикла: RB-свойства выполнены всюду.

правым. Если правым, исполняются строки 12–13 (случай 2). В этом случае выполняется левое вращение, которое сводит случай 2 к случаю 3. когда x является левым ребёнком (рис. 14.6), Так как и x, и $p[x]$ красные, после вращения количества чёрных вершин на путях остаются прежними.

Итак, осталось рассмотреть случай 3: красная вершина x является левым ребёнком красной вершины $p[x]$, которая является левым ребёнком чёрной вершины $p[p[x]]$, правым ребёнком которой является чёрная вершина y. В этом случае достаточно произвести правое вращение и перекрасить две вершины, чтобы устранить нарушение RB-свойств. Цикл больше не выполняется, так как вершина $p[x]$ теперь чёрная.

Каково время выполнения процедуры RB-Insert? Высота красно-чёрного дерева есть $O(\lg n)$, если в дереве n вершин, поэтому вызов Tree-Insert требует времени $O(\lg n)$. Цикл повторяется, только если мы встречаем случай 1, и при этом x сдвигается вверх по дереву. Таким образом, цикл повторяется $O(\lg n)$ раз, и общее время работы есть $O(\lg n)$. Интересно, что при этом выполняется не более двух вращений (после которых мы выходим из цикла).

Упражнения

14.3-1 В строке 2 процедуры RB-Insert мы красим новую вершину x в красный цвет. Если бы мы покрасили её в черный цвет, свойство 3 не было бы нарушено. Почему же мы этого не сделали?

14.3-2 В строке 18 мы красим корень дерева в чёрный цвет. Зачем это делается?

14.3-3 Нарисуйте красно-чёрные деревья, которое получаются при последовательном добавлении к пустому дереву ключей

$41, 38, 31, 12, 19, 8$.

14.3-4 Пусть чёрная высота каждого из поддеревьев α, β, γ, δ, ε на рисунках 14.5 и 14.6 равна k. Найдите чёрные высоты всех вершин на этих рисунках и проверьте, что свойство 4 действительно не нарушается.

14.3-5 Рассмотрим красно-чёрное дерево, полученное добавлением n вершин к пустому дереву. Убедитесь, что при $n > 1$ в дереве есть хотя бы одна красная вершина.

14.3-6 Как эффективно реализовать процедуру RB-INSERT, если в вершинах не хранятся указатели на родителей?

14.4 Удаление

Как и другие операции, удаление вершины из красно-чёрного дерева требует времени $O(\lg n)$. Удаление вершины несколько сложнее вставки.

Чтобы упростить обработку граничных условий, мы используем фиктивный элемент (по-английски называемый sentinel) вместо NIL (см. с. ??). Для красно-чёрного дерева T фиктивный элемент $nil[T]$ имеет те же поля, что и обычная вершина дерева. Его цвет чёрный, а остальным полям (p, *left*, *right* и *key*) могут быть присвоены любые значения. Мы считаем, что в красно-чёрном дереве все указатели NIL заменены указателями на $nil[T]$.

Благодаря фиктивным элементам мы можем считать NIL-лист, являющийся ребёнком вершины x, обычной вершиной, родитель которой есть x. В принципе можно было бы завести по одной фиктивной вершине для каждого листа, но это было бы напрасной потерей памяти. Чтобы избежать этого, мы используем один элемент $nil[T]$, представляющий все листы. Однако, когда мы хотим работать с листом — ребёнком вершины x, надо не забыть выполнить присваивание $p[nil[T]] \leftarrow x$.

Процедура RB-DELETE следует схеме процедуры TREE-DELETE из раздела 13.3. Вырезав вершину, она вызывает вспомогательную процедуру RB-DELETE-FIXUP, которая меняет цвета и производит вращения, чтобы восстановить RB-свойства.

```
RB-Delete$(T, z)$
 1  if $left[z] = nil[T]$ или $right[z] = nil[T]$
 2     then $y \leftarrow z$
 3     else $y \leftarrow$ Tree-Successor$(z)$
 4  if $left[y] \neq nil[T]$
 5     then $x \leftarrow left[y]$
 6     else $x \leftarrow right[y]$
 7  $p[x] \leftarrow p[y]$
 8  if $p[y] = nil[T]$
 9     then $root[T] \leftarrow x$
10     else if $y = left[p[y]]$
11             then $left[p[y]] \leftarrow x$
12             else $right[p[y]] \leftarrow x$
13  if $y \neq z$
14     then $key[z] \leftarrow key[y]$
15          ▷ Копируем дополнительные данные из вершины $y$.
16  if $color[y] =$ BLACK
17     then RB-Delete-Fixup$(T, x)$
18  return $y$
```

Есть три различия между процедурами RB-Delete и Tree-Delete. Во-первых, вместо NIL всюду стоит указатель на фиктивный элемент *nil*$[T]$. Во-вторых, проверка $x \neq$ NIL в строке 7 процедуры Tree-Delete удалена, и присваивание $p[x] \leftarrow p[y]$ выполняется в любом случае. Если x есть фиктивный элемент *nil*$[T]$, то его указатель на родителя становится равным родителю удаляемого элемента y. В-третьих, в строках 16–17 вызывается процедура RB-Delete-Fixup, если удаляемая вершина y — чёрная. При удалении красной вершины RB-свойства не нарушаются (чёрные высоты не меняются, и красные вершины не могут стать соседними). Передаваемая процедуре RB-Delete-Fixup вершина x являлась единственным ребёнком вершины y, если у y был ребёнок (не являющийся листом), или фиктивным элементом *nil*$[T]$, если вершина y не имела детей. В последнем случае присваивание в строке 7 гарантирует, что $p[x]$ указывает на бывшего родителя y — вне зависимости от того, является ли x настоящей вершиной или фиктивным элементом.

Посмотрим, как процедура RB-Delete-Fixup восстанавливает RB-свойства дерева.

```
RB-Delete-Fixup($T, x$)
 1  while $x \neq root[T]$ и $color[x] =$ BLACK
 2      do if $x = left[p[x]]$
 3            then $w \leftarrow right[p[x]]$
 4                 if $color[w] =$ RED
 5                    then $color[w] \leftarrow$ BLACK               ▷ Случай 1
 6                         $color[p[x]] \leftarrow$ RED              ▷ Случай 1
 7                         Left-Rotate($T, p[x]$)                    ▷ Случай 1
 8                         $w \leftarrow right[p[x]]$                ▷ Случай 1
 9                 if $color[left[w]] =$ BLACK и
                         $color[right[w]] =$ BLACK
10                    then $color[w] \leftarrow$ RED                 ▷ Случай 2
11                         $x \leftarrow p[x]$                       ▷ Случай 2
12                    else if $color[right[w]] =$ BLACK
13                            then $color[left[w]] \leftarrow$ BLACK ▷ Случай 3
14                                 $color[w] \leftarrow$ RED         ▷ Случай 3
15                                 Right-Rotate($T, w$)              ▷ Случай 3
16                                 $w \leftarrow right[p[x]]$        ▷ Случай 3
17                         $color[w] \leftarrow color[p[x]]$         ▷ Случай 4
18                         $color[p[x]] \leftarrow$ BLACK            ▷ Случай 4
19                         $color[right[w]] \leftarrow$ BLACK        ▷ Случай 4
20                         Left-Rotate($T, p[x]$)                    ▷ Случай 4
21                         $x \leftarrow root[T]$                    ▷ Случай 4
22            else (симметричный фрагмент с
                 заменой $left \leftrightarrow right$)
23  $color[x] \leftarrow$ BLACK
```

Если удалённая процедурой RB-Delete вершина y была чёрной, то любой путь, через неё проходивший, теперь содержит на одну чёрную вершину меньше. Таким образом, свойство 4 нарушилось. Мы можем компенсировать это за счёт вершины x (занявшей место вершины y). Если x — красная, сделаем её чёрной (заодно мы избегаем опасности получить красную вершину с красным родителем). Если x — чёрная, объявим её *"дважды чёрной"* и будем считать за две при подсчёте числа чёрных вершин на пути от корня к листьям. Конечно, такой выход может быть лишь временным, поскольку определение красно-чёрных деревьев не предусматривает дважды чёрных вершин, и мы должны постепенно от такой вершины избавиться.

Процедура RB-Delete-Fixup(T, x) применяется к дереву, которое *обладает свойствами красно-чёрного дерева, если учесть дополнительную единицу черноты в вершине* x, и превращает его в настоящее красно-чёрное дерево. В цикле (строки 1–22) дерево меняется, и значение переменной x тоже меняется (выделенная вершина может сдвигаться вверх по дереву), но сформулированное свойство остаётся верным.

Цикл завершается, если (1) x указывает на красную вершину (тогда мы в строке 23 красим её в чёрный цвет) или если (2) x указывает на корень (тогда лишняя чернота может быть просто удалена из дерева). Может оказаться также, что (3) внутри тела цикла удаётся выполнить несколько вращений и перекрасить несколько вершин, после чего дважды чёрная вершина исчезнет. В этом случае присваивание $x \leftarrow root[T]$ позволяет выйти из цикла.

Внутри цикла x указывает на дважды чёрную вершину, не являющуюся корнем. В строке 2 мы определяем, каким ребёнком является x — левым или правым. (Подробно выписана часть процедуры для первого случая, второй случай симметричен и скрыт в строке 22.) Переменная w (строка 3) указывает на второго ребёнка вершины $p[x]$ (*"брата"* вершины x). Так как вершина x — дважды чёрная, w не может быть равно $nil[T]$, поскольку в этом случае вдоль одного пути от $p[x]$ вниз (через w) было бы меньше черных вершин, чем вдоль другого (через x).

Четыре возможных случая показаны на рис. 14.7. Прежде чем разбираться с ними детально, посмотрим, как проверить, что преобразования не нарушают свойство 4. Достаточно убедиться, что количество чёрных вершин от корня показанного поддерева до каждого из поддеревьев $\alpha, \beta, \ldots, \zeta$ не изменилось. Например, на рис. 14.7а, иллюстрирующем случай 1, количество чёрных вершин от корня до каждого из поддеревьев α и β равно 3 как до, так и после преобразования. (Напомним, что вершина x считается за две.) Аналогично, количество чёрных вершин от корня до γ, δ, ε и ζ, равно 2 до и после преобразования. На рис. 14.7б вершина B может быть и чёрной, и красной. Если она красная, то число чёрных вершин от корня до α (до и после преобразования) равно 2, если чёрная — то 3. Остальные случаи проверяются аналогично (упр. 14.4-5).

Итак, рассмотрим все случаи по порядку. Случай 1 (строки 5–8 процедуры RB-DELETE-FIXUP, рис. 14.7а) имеет место, когда вершина w, брат x, красная (в этом случае их родитель, $p[x]$, чёрный). Так как оба ребёнка вершины w чёрные, мы можем поменять цвета w и $p[x]$ и произвести левое вращение вокруг $p[x]$, не нарушая RB-свойств. Вершина x остаётся дважды чёрной, а её новый брат — чёрный, так что мы свели дело к одному из случаев 2, 3 или 4.

Если вершина w чёрная, имеет место один из случаев 2–4. Они различаются между собой цветом детей вершины w. В случае 2 (строки 10–11, рис. 14.7б) оба ребёнка вершины w чёрные. Так как вершина w тоже чёрная, мы можем снять чёрную окраску с x (лишнюю) и с w (сделав её красной), и добавить черноту родителю, $p[x]$. После этого продолжим выполнение цикла. Заметим, что если мы попали в случай 2 из случая 1, то вершина $p[x]$ — красная, поэтому цикл сразу же завершится (добавив чёрного к красной вершине, мы красим её в обычный чёрный цвет).

В случае 3 (строки 13–16, рис. 14.7в) вершина w чёрная, её левый ребёнок — красный, а правый — чёрный. Мы можем поменять цвета w и её левого ребёнка и потом применить правое вращение так, что RB-свойства будут сохранены. Новым братом вершины x теперь будет чёрная вершина с красным правым ребёнком, и мы свели случай 3 к случаю 4.

Наконец, в случае 4 (строки 17–21, рис. 17.4г) вершина w (брат вершины x) является чёрной, а её правый ребёнок — красный. Меняя некоторые цвета и производя левое вращение вокруг $p[x]$, мы можем удалить излишнюю черноту у x, не нарушая RB-свойств. Присваивание $x \leftarrow root[T]$ выводит нас из цикла.

Каково время выполнения процедуры RB-DELETE? Высота красно-чёрного дерева с n вершинами есть $O(\lg n)$, поэтому время исполнения RB-DELETE без учёта RB-DELETE-FIXUP есть $O(\lg n)$. Сколько времени требует цикл в процедуре RB-DELETE-FIXUP? Как только обнаруживается случай 1, 3 или 4, мы выходим из цикла (при этом выполняется $O(1)$ операций и самое большее три вращения). До этого возможно несколько повторений случая 2, но при каждом повторении указатель x перемещается вверх по дереву и никакие вращения не производятся, так что число таких шагов есть $O(\lg n)$. Таким образом, процедура RB-DELETE-FIXUP требует времени $O(\lg n)$, и общее время работы процедуры RB-DELETE также есть $O(\lg n)$ (отметим ещё раз, что при этом производится не более трёх вращений).

Упражнения

14.4-1 Убедитесь, что после выполнения процедуры RB-DELETE корень дерева остаётся чёрным, если он таковым был.

14.4-2 В упр. 14.3-3 построено красно-чёрное дерево, которое получается добавлением ключей $41, 38, 31, 12, 19, 8$ в пустое дерево. Нарисуйте деревья, которые получатся из него при последовательном удалении ключей $8, 12, 19, 31, 38, 41$.

14.4-3 В каких строках процедуры RB-DELETE-FIXUP мы можем читать или изменять фиктивный элемент $nil[T]$?

14.4-4 Упростите процедуру LEFT-ROTATE, используя фиктивный элемент для представления NIL и ещё один фиктивный элемент, содержащий указатель на корень дерева.

14.4-5 Для каждого из случаев на рисунке 14.7 подсчитайте количество чёрных вершин от корня поддерева на рисунке до каждого из поддеревьев $\alpha, \beta, \ldots, \zeta$ и убедитесь, что оно не меняется при преобразованиях. Используйте обозначение $count(c)$ для *"степени чер-*

Рисунок 14.7 Четыре случая, возможных в основном цикле процедуры RB-Delete-Fixup. Чёрные вершины показаны как чёрные, красные показаны тёмно-серыми. Светло-серые вершины на рисунке могут быть и красными, и чёрными. (Их цвета обозначаются c и c'.) Буквы $\alpha, \beta, \ldots, \zeta$ обозначают произвольные поддеревья. В каждом случае конфигурация слева преобразуется в конфигурацию справа перекрашиванием вершин и/или вращениями. Вершина, на которую указывает x, дважды чёрная. Единственный случай, когда выполнение цикла продолжается — случай 2. (а) Случай 1 сводится к случаю 2, 3 или 4, если поменять местами цвета вершин B и D и произвести левое вращение. (б) В случае 2 *"избыток черноты"* в вершине x перемещается вверх по дереву, когда мы делаем D красной и устанавливаем указатель x в B. Если мы попали в случай 2 из случая 1, то цикл завершается, так как вершина B была красной. (в) Случай 3 сводится к случаю 4, если поменять местами цвета вершин C и D и выполнить правое вращение. (г) В случае 4 можно перекрасить некоторые вершины и выполнить левое вращение (не нарушив RB-свойства) так, что лишний чёрный цвет исчезает, и цикл можно завершить.

ноты" вершины цвета c, считая $count(c)$ равным 0 для красной вершины и 1 для чёрной.

14.4-6 Предположим, что вершина вставлена в красно-чёрное дерево, а потом сразу же удалена. Будет ли получившееся дерево совпадать с исходным? Почему?

Задачи

14-1 Динамические множества с сохранением предыдущих версий

Иногда полезно сохранять предыдущие версии меняющегося множества. (Такие структуры данных называются по-английски persistent data structures.) Можно, конечно, копировать множество каждый раз, когда оно изменяется. Но такой подход требует много памяти и времени — и есть способы, позволяющие сделать это более эффективно.

Мы хотим предусмотреть возможность хранения предыдущих версий для множества S с операциями INSERT, DELETE и SEARCH. Мы считаем, что множество S реализовано с помощью двоичных деревьев поиска, как показано на рис. 14.8а. Для каждой версии множества мы храним свой отдельный корень. Чтобы добавить ключ 5, мы создаём новую вершину с этим ключом. Эта вершина становится левым ребёнком новой вершины с ключом 7, так как существующую вершину менять нельзя. Подобным образом новая вершина с ключом 7 становится левым ребёнком новой вершины с ключом 8, правый ребёнок которой — существующая вершина с ключом 10. В свою очередь, новая вершина с ключом 8 становится правым ребёнком нового корня r' с ключом 4, левый ребёнок которого — существующая вершина с ключом 3. Таким образом, мы копируем лишь часть дерева, а в остальном используем старое, как это показано на рис. 14.8б.

Мы предполагаем, что вершины дерева содержат поля key, $left$ и $right$, но не содержат поля p, указывающего на родителя. (См. также упр. 14.3-6.)

а Покажите, какие вершины хранимого таким образом дерева должны быть изменены (созданы) в общем случае при добавлении или удалении элемента.

б Напишите процедуру PERSISTENT-TREE-INSERT, которая добавляет ключ k в дерево T.

в Если высота дерева равна h, сколько времени и памяти требует написанная вами процедура? (Количество памяти можно измерять количеством новых вершин.)

г Пусть мы используем и поля p в вершинах дерева. В этом случае процедура PERSISTENT-TREE-INSERT должна будет выполнить

Рисунок 14.8 (а) Двоичное дерево поиска с ключами $2, 3, 4, 7, 8, 10$. (б) Дерево с сохранением предыдущих версий после добавления ключа 5. Текущая версия состоит из вершин, доступных из текущего корня r', а предыдущая версия содержит вершины, доступные из старого корня r. Тёмно-серые вершины добавлены при добавлении ключа 5.

дополнительные действия. Покажите, что в этом случае время работы и объём необходимой памяти будут $\Omega(n)$, где n — количество вершин в дереве.

д Покажите, как можно использовать красно-чёрные деревья, чтобы гарантировать, что добавление и удаление элемента для множества с хранением предыдущих версий будут требовать времени $O(\lg n)$ в худшем случае.

14-2 Операция объединения красно-чёрных деревьев

Операция **объединения** (join) применяется к двум динамическим множествам S_1 и S_2 и элементу x, причём заранее известно, что $key[x_1] \leqslant key[x] \leqslant key[x_2]$ для любых $x_1 \in S_1$ и $x_2 \in S_2$. Её результатом является множество $S = S_1 \cup \{x\} \cup S_2$. В этой задаче мы покажем, как реализовать операцию объединения для красно-чёрных деревьев.

а Мы будем хранить чёрную высоту красно-чёрного дерева T в специальной переменной $bh[T]$. Убедитесь, что это значение можно поддерживать, не размещая никакой дополнительной информации в вершинах дерева и не ухудшая асимптотику времени работы процедур RB-Insert и RB-Delete. Покажите, что, спускаясь по дереву, можно вычислить чёрную высоту каждой вершины за время $O(1)$ в расчёте на каждую просмотренную вершину.

Мы хотим реализовать операцию RB-Join(T_1, x, T_2), которая из двух деревьев T_1 и T_2 формирует новое красно-чёрное дерево $T = T_1 \cup \{x\} \cup T_2$ (старые деревья при этом разрушаются). Пусть n — общее количество вершин в T_1 и T_2.

б Считая, что $bh[T_1] \geqslant bh[T_2]$, опишите алгоритм со временем работы $O(\lg n)$, находящий среди чёрных вершин дерева T_1, имеющих чёрную высоту $bh[T_2]$, вершину y с наибольшим ключом.

в Пусть T_y — поддерево с корнем y. Покажите, как заменить T_y на $T_y \cup \{x\} \cup T_2$ за время $O(1)$ без потери свойства упорядоченности.

г В какой цвет надо покрасить x, чтобы сохранить RB-свойства 1, 2 и 4? Объясните, как восстановить свойство 3 за время $O(\lg n)$.

д Убедитесь, что время выполнения процедуры RB-Join есть $O(\lg n)$.

Замечания

Идея балансировки двоичных деревьев поиска принадлежит Г. М. Адельсону-Вельскому и Е. М. Ландису [2], предложившим в 1962 году класс сбалансированных деревьев, называемых теперь АВЛ-деревьями. Баланс поддерживается с помощью вращений; для его восстановления после добавления или удаления вершины может потребоваться $\Theta(\lg n)$ вращений (для дерева с n вершинами). Ещё один класс деревьев поиска, называемых 2-3-деревьями, был предложен Хопкрофтом (J. E. Hopcroft, не опубликовано) в 1970 году. Здесь баланс поддерживается за счёт изменения степеней вершин. Обобщение 2-3-деревьев, называемое Б-деревьями, предложили Байер и МакКрейт [18]; Б-деревья обсуждаются в главе 19.

Красно-чёрные деревья предложил Байер [17], назвав их *"симметричными двоичными Б-деревьями"*. Гибас и Седжвик [93] подробно изучили их свойства и предложили использовать для наглядности красный и чёрный цвета.

Из многих других вариаций на тему сбалансированных деревьев наиболее любопытны, видимо, *"расширяющиеся деревья"* (splay trees), которые придумали Слеатор и Тарьян [177]. Эти деревья являются *" саморегулирующимися"*. (Хорошее описание расширяющихся деревьев дал Тарьян [188].) Расширяющиеся деревья поддерживают баланс без использования дополнительных полей (типа цвета). Вместо этого *"расширяющие операции"* (splay operations), включающие вращения, выполняются при каждом обращении к дереву. Учётная стоимость (amortized cost, гл. 18) в расчёте на одну операцию с деревом для расширяющихся деревьев составляет $O(\lg n)$.

15 Пополнение структур данных

Далеко не во всех ситуациях можно обойтись лишь классическими структурами данных (двоичными деревьями поиска, двусторонне связанными списками, хеш-таблицами и т. п.) Однако редко требуется придумать что-то совсем новое: в большинстве случаев достаточно расширить какую-либо из классических структур данных, храня вместе с её объектами дополнительную информацию. Эффективное обновление этой информации при выполнении операций иногда требует немалой изобретательности.

В качестве примера в этом разделе рассматриваются красно-чёрные деревья. Храня в вершинах дополнительную информацию, мы сможем быстро находить i-й по порядку элемент, а также выполнять обратное действие: находить порядковый номер данного элемента множества (разд. 15.1). В разделе 15.2 обсуждается общая схема работы с дополнительной информацией и доказывается теорема, которая облегчает пополнение для красно-чёрных деревьев. В разделе 15.3 эта же теорема используется для построения структуры данных, хранящей динамическое множество промежутков (на числовой прямой) и позволяющей быстро находить элемент множества, перекрывающийся с заданным промежутком.

15.1 Динамические порядковые статистики

Порядковые статистики уже обсуждались в главе 10, где мы искали i-й по порядку элемент множества из n элементов, что требовало $O(n)$ операций (если множество предварительно не упорядочено). В данном разделе мы покажем, как с помощью пополненных красно-чёрных деревьев найти i-й элемент за $O(\log n)$ операций. Кроме того, за то же время можно будет найти порядковый номер заданного элемента.

Подходящая структура данных показана на рис. 15.1. **Порядковым деревом** (order-statistic tree) мы называем красно-чёрное дерево T, каждая вершина x которого, помимо обычных полей $key[x]$, $color[x]$, $p[x]$, $left[x]$ и $right[x]$ имеет поле $size[x]$. В нём хранится размер (количество вершин, не считая NIL-листьев) поддерева с корнем в x (считая и саму вершину x). Считая, что в поле *size*[NIL] записан 0, можно написать такое соотношение:

$$size[x] = size[left[x]] + size[right[x]] + 1.$$

(При реализации можно использовать фиктивный элемент *nil*[T], как в разделе 14.4, а можно каждый раз проверять, не равен ли указатель значению NIL, и подставлять 0 вместо значения поля *size*.)

Поиск i-го по величине элемента

Мы должны уметь обновлять дополнительную информацию (поля *size*) при добавлении и удалении элементов. Но сначала объясним, как ею пользоваться. Начнём с поиска i-го элемента. Рекурсивная процедура OS-SELECT(x, i) возвращает указатель на i-й элемент поддерева с корнем x. Найти i-й элемент во всём дереве T можно с помощью вызова OS-SELECT$(root[T], i)$.

```
OS-SELECT($x, i$)
1  $r \leftarrow size[left[x]]+1$
2  if $i = r$
3     then return $x$
4  elseif $i < r$
5     then return OS-SELECT($left[x], i$)
6  else return OS-SELECT($right[x], i - r$)
```

Этот алгоритм использует ту же идею, что и алгоритмы поиска главы 10. В поддереве с корнем x сначала идут *size*[*left*[x]] вершин левого поддерева, меньших x, затем сама вершина x (которая является $(size[left[x]] + 1)$-й по счёту), и затем вершины правого поддерева x.

Процедура начинает с вычисления порядкового номера r вершины x (строка 1). Если $i = r$ (строка 2), то x и есть i-й элемент, и мы возвращаем его (строка 3). Если $i < r$, то искомый элемент находится в левом поддереве вершины x, и программа рекурсивно вызывает себя (строка 5). Если же $i > r$, то искомый элемент находится в правом поддереве вершины x, но его порядковый номер внутри этого поддерева будет уже не i, а $i - r$. Этот элемент выдаётся при рекурсивном вызове в строке 6.

Покажем, как процедура OS-SELECT ищет 17-й элемент дерева, изображённого на рис. 15.1. Мы начинаем с корня (с ключом 26), при этом $i = 17$. Размер левого поддерева корня равен 12, поэтому порядковый номер корня равен 13, и искомая вершина находится в правом поддереве, имея там порядковый номер $17 - 13 = 4$. Ищем 4-й элемент поддерева с корнем 41. Левое поддерево вершины 41 имеет размер 5, т.е. порядковый номер вершины 41 равен 6. Ищем

Рисунок 15.1 Порядковое дерево. Красные вершины изображены как светло-серые, чёрные — как тёмные. Помимо обычных полей красно-чёрного дерева, у каждой вершины x есть поле $size[x]$, где хранится размер поддерева с корнем в x.

4-й элемент поддерева с корнем 30. В этом дереве корень имеет порядковый номер 2, поэтому ищем $(4 - 2) = 2$-й элемент правого поддерева. В этом дереве корень 38 оказался вторым по порядку (размер левого поддерева равен 1), поэтому мы возвращаем указатель на вершину 38.

При каждом рекурсивном вызове процедуры мы спускаемся по дереву на один уровень, поэтому время работы в худшем случае пропорционально высоте. Высота красно-чёрного дерева с n вершинами равна $O(\log n)$, так что время работы процедуры OS-Select на n-элементном множестве есть $O(\log n)$.

Определение порядкового номера элемента

Процедура OS-Rank получает указатель на элемент x порядкового дерева T и находит порядковый номер (rank) элемента в дереве.

```
OS-Rank($T, x$)
1  $r \leftarrow size[left[x]] + 1$
2  $y \leftarrow x$
3  while $y \neq root[T]$
4      do if $y = right[p[y]]$
5            then $r \leftarrow r + size[left[p[y]]] + 1$
6         $y \leftarrow p[y]$
7  return $r$
```

Порядковый номер элемента x на единицу больше количества элементов, меньших x. Цикл в строках 3–6 имеет такой инвариант: r есть порядковый номер вершины x в поддереве с корнем y. Перед первым выполнением цикла это так: r есть порядковый номер x в дереве с корнем в x (строка 1), а $y = x$ (строка 2). Как надо изменить r при переходе от вершины y к её родителю $p[y]$? Если y — левый ребёнок вершины $p[y]$, то вершина x имеет один и тот же порядковый номер в поддеревьях с корнями в y и $p[y]$. Если же y — правый ребёнок, при переходе от y к $p[y]$ к порядковому номеру вершины x прибавится $size[left[p[y]]]$ элементов левого поддерева и сама вершина $p[y]$. В строке 5 мы учли эту добавку.

При выходе из цикла $y = root[T]$, а r — порядковый номер x во всём дереве.

Для примера посмотрим, как эта процедура определит порядковый номер вершины с ключом 38 в дереве, изображённом на рис. 15.1. В таблице показан ключ вершины y и число r в начале каждого цикла.

Шаг	$key[x]$	r
1	38	2
2	30	4
3	41	4
4	26	17

Возвращается значение 17.

С каждым шагом глубина вершины y уменьшается на единицу, и каждый шаг требует времени $O(1)$, поэтому время работы процедуры OS-RANK на n-элементном множестве есть $O(\log n)$.

Обновление информации о размерах поддеревьев

С помощью полей $size$ мы научились быстро вычислять порядковый номер элемента, а также находить i-й элемент дерева. Теперь надо понять, как обновлять это поле при добавлении и удалении элемента из красно-чёрного дерева. Оказывается, это можно сделать, не ухудшив асимптотическую оценку времени работы процедур добавления и удаления элемента.

Как мы видели в разделе 14.3, добавление элемента в красно-чёрное дерево состоит из двух частей: сначала мы добавляем новую вершину, делая её ребёнком уже существующей, а затем производим некоторые вращения и перекрашивания (если это необходимо)

На первом этапе мы для каждой пройденной вершины x (на пути от корня до места добавления вершины) увеличим на единицу значение $size[x]$. В поле $size$ добавленной вершины запишем число 1. При этом мы проходим путь длиной $O(\log n)$ и дополнительно тратим время $O(\log n)$, после чего все поля $size$ правильны.

При вращении изменяются размеры только двух поддеревьев: их вершинами являются концы ребра, вокруг которого происходило вращение. Поэтому к тексту процедуры LEFT-ROTATE(T, x) (раздел 14.2) достаточно добавить строки

```
13  $size[y] \leftarrow size[x]$
14  $size[x] \leftarrow size[left[x]] + size[right[x]] + 1$
```

Рисунок 15.2 Обновление поля *size* при вращениях вокруг ребра (x, y). Это поле необходимо обновить только у вершин x и y. Для того, чтобы узнать новое значение, достаточно обладать информацией, хранящейся в x, y и в корнях поддеревьев, изображенных в виде треугольников.

На рис. 15.2 показано, как меняется поле *size*. Аналогичные изменения внесём и в процедуру RIGHT-ROTATE.

Так как при добавлении элемента в красно-чёрное дерево выполняется не больше двух вращений, то коррекция поля *size* после вращений требует $O(1)$ операций (а перекрашивание вообще не влияет на размеры). Таким образом, добавление элемента в порядковое дерево из n вершин требует времени $O(\log n)$, как и для обычного красно-чёрного дерева.

Удаление элемента из красно-чёрного дерева также состоит из двух частей — сначала мы удаляем вершину y, а затем делаем (самое большее) три вращения (см. разд. 14.4). После выполнения первой части, мы уменьшим на единицу поля *size* у всех вершин на пути из y к корню. Длина этого пути $O(\log n)$, поэтому дополнительно нам потребуется время $O(\log n)$. Что происходит с вращениями, мы уже видели. Таким образом, обновление поля *size*, не ухудшает (асимптотически) время, необходимое для удаления и добавления элемента.

Упражнения

15.1-1 Пусть T — дерево на рис. 15.1. Покажите, как работает процедура OS-SELECT$(T, 10)$.

15.1-2 Пусть T — дерево на рис. 15.1. Покажите, как работает процедура OS-RANK(T, x), где x — вершина с ключом 35.

15.1-3 Перепишите процедуру OS-SELECT, не используя рекурсии.

15.1-4 Реализуйте рекурсивную процедуру OS-KEY-RANK(T, k), которая получает на входе порядковое дерево T с попарно различными ключами, а также ключ k, и возвращает порядковый номер ключа k в этом дереве.

15.1-5 Дана вершина x порядкового дерева из n вершин и натуральное число i. Как найти i-й по порядку элемент, считая от вершины x, за время $O(\log n)$?

15.1-6 Заметим, что процедуры OS-Select и OS-Rank используют поле *size* только для того, чтобы узнать порядковый номер вершины x в поддереве с корнем x. Предположим, что мы храним в вершине вместо поля *size* этот порядковый номер. Как обновлять эту информацию при добавлении и удалении элемента? (Напомним, что добавление и удаление сопровождаются вращениями.)

15.1-7 Как, используя порядковые деревья, посчитать число инверсий (см. задачу 1-3) в массиве размера n за время $O(n \log n)$?

15.1-8* Рассмотрим n хорд окружности, заданных своими концами (считаем, что все $2n$ концевых точек различны). Придумайте алгоритм, который за время $O(n \lg n)$ определяет, сколько пар хорд пересекаются внутри круга. (Например, если все хорды — диаметры, то ответом будет C_n^2.)

15.2 Общая схема работы с дополнительной информацией

Ситуация, когда требуется пополнить какую-либо стандартную структуру данных дополнительной информацией, довольно типична. Мы встретимся с ней снова в следующем разделе. А в этом разделе мы докажем общую теорему, облегчающую этот процесс в случае красно-чёрных деревьев.

Пополнение структуры данных делится на четыре шага:

1. выбираем базовую структуру данных;
2. решаем, какую дополнительную информацию мы будем хранить (и обновлять);
3. проверяем, что эту информацию удаётся обновлять при выполнении операций, допустимых для выбранной структуры данных;
4. реализуем новые операции.

Конечно, эти правила — всего лишь общая схема, и в конкретной ситуации надо проявлять разумную гибкость, но схема эта может быть полезной.

Давайте посмотрим на конструкции раздела 15.1 с точки зрения этих правил.

На шаге 1 мы выбрали красно-чёрные деревья в качестве базовой структуры.

На шаге 2 мы решили добавить к каждой вершине поле *size*. Смысл хранения дополнительной информации состоит в том, что она позволяет выполнять некоторые операции быстрее. Без поля *size* мы не смогли бы выполнить операции OS-Select и OS-Rank за время $O(\log n)$. (Несколько иной вариант выбора дополнительной информации дан в упр. 15.2-1.)

На шаге 3 мы убедились, что обновление полей *size* при добавлении и удалении элементов можно выполнить без ухудшения асимптотики для времени добавления и удаления. В этом смысле наш выбор удачен: если бы, скажем, мы решили хранить для каждой вершины её порядковый номер, то процедуры OS-SELECT и OS-RANK работали бы быстро, но добавление нового элемента повлекло бы за собой изменение дополнительной информации во многих вершинах дерева (во всех, если добавленный элемент минимален). Наш выбор (поле *size*) даёт удачный компромисс между лёгкостью использования и обновления.

На шаге 4 мы реализовали процедуры OS-SELECT и OS-RANK, из-за которых мы, собственно, и затевали всё это дело.

Впрочем, в других ситуациях (упр. 15.2-1) дополнительная информация используется не для реализации новых операций, а для ускорения уже имеющихся.

Дополнительная информация для красно-чёрных деревьев

Следующая теорема показывает, что для красно-чёрных деревьев информацию определённого вида можно обновлять, не замедляя (асимптотически) операции добавления и удаления. Её доказательство во многом повторяет рассуждения из раздела 15.1.

Теорема 15.1 (Пополнение красно-чёрного дерева). *Рассмотрим дополнительный атрибут f, определённый для вершин красно-чёрных деревьев. Предположим, что для всякой вершины x значение $f[x]$ полностью задаётся остальной информацией, хранящейся в вершинах x, $left[x]$ и $right[x]$ (в том числе значениями $f[left[x]]$ и $f[right[x]]$), и его вычисление по этим данным требует времени $O(1)$. Тогда поля f можно обновлять при добавлении и удалении элемента из дерева, не ухудшая (асимптотически) время выполнения добавления и удаления.*

Доказательство. Идея доказательства состоит в том, что изменение поля f в некоторой вершине x повлечёт за собой изменения поля f только в вершинах, расположенных на пути из корня в x. В самом деле, изменение $f[x]$ повлечёт за собой изменение $f[p[x]]$, что в свою очередь изменит $f[p[p[x]]]$ и т.д., но другие вершины останутся нетронутыми. От $f[root[T]]$ не зависит значение поля f в других вершинах, и процесс изменений остановится. Так как высота дерева равна $O(\log n)$, то после изменения поля $f[x]$ для какой-то одной вершины x мы сможем обновить все необходимые поля за время $O(\log n)$.

Добавление элемента x в дерево T делается в два этапа (см. разд. 14.3). На первом этапе вершину x добавляют в качестве ребёнка уже существующей вершины $p[x]$. Значение $f[x]$ вычисляется за время $O(1)$, так как новая вершина — лист (точнее, её дети — NIL-листья). Далее происходит $O(\lg n)$ изменений полей на пути к корню. Итак, первый этап занимает время $O(\log n)$. На втором этапе мы выполняем вращения (самое большее два); после каждого потребуется $O(\lg n)$ операций для распространения изменений вверх по дереву.

Удаление также проводится в два этапа (см. разд. 14.4). На первом этапе изменения возникают, если удаляемая вершина заменяется её последователем, а также когда мы выбрасываем удаляемую вершину или её последователя. И в том и в другом случае мы изменяем поле f у одной вершины, поэтому обновление всех полей займет время $O(\log n)$. На втором этапе мы делаем самое большее три вращения, каждое из которых требует времени $O(\log n)$ для обновления полей на пути к корню. ☐

Во многих случаях (в частности, для поля *size*) при вращениях все поля можно обновить за время $O(1)$, а не $O(\log n)$. Такая ситуация возникает в упр. 15.2-4.

Упражнения

15.2-1 Пополнить порядковое дерево (не ухудшив асимптотически время операций) так, чтобы минимальный и максимальный элементы, а также предшественник и последователь данного элемента отыскивались бы за время $O(1)$.

15.2-2 Будем хранить в каждой вершине красно-чёрного дерева её чёрную высоту. Возможно ли обновлять это поле при добавлении и удалении элемента из дерева, не ухудшив (асимптотически) время работы этих операций?

15.2-3 Будем хранить в вершине её глубину. Возможно ли обновлять это поле при добавлении и удалении элемента из дерева, не ухудшив (асимптотически) время работы этих операций?

15.2-4* Пусть $\otimes$ — ассоциативная бинарная операция на некотором множестве M, и пусть в каждой вершине красно-чёрного дерева хранится некоторый элемент a множества M (свой в каждой вершине). Пусть теперь мы хотим хранить в каждой вершине x поле $f[x] = a[x_1] \otimes a[x_2] \otimes \ldots \otimes a[x_m]$, где $x_1, x_2, \ldots, x_m$ — все вершины поддерева с корнем x (в порядке возрастания ключей). Показать, что поле f при вращениях можно обновлять за время $O(1)$. Провести аналогичное рассуждение для поля *size*.

15.2-5* Мы хотим реализовать для красно-чёрных деревьев операцию RB-ENUMERATE(x, a, b), которая выдаёт список всех вершин в поддереве с корнем x, для которых ключ k находится в промежутке $a \leqslant k \leqslant b$. Как сделать это за время $\Theta(m + \log n)$, где n — количество вершин в дереве, а m — число выдаваемых ключей? (Указание. Достаточно полей, имеющихся в красно-чёрных деревьях; новые поля не нужны.)

15.3 Деревья промежутков

В этом разделе мы используем красно-чёрные деревья для хранения меняющегося множества промежутков. **Отрезком** (closed interval) $[t_1, t_2]$ называется множество вещественных чисел t, для которых $t_1 \leqslant t \leqslant t_2$. (Предполагается, что $t_1 \leqslant t_2$.) **Полуинтервал** (half-open interval) и **интервал** (open interval) получаются из отрезка выкидыванием одного или двух концов соответственно. В этом разделе мы имеем дело только с отрезками, но все результаты легко распространяются на интервалы и полуинтервалы.

Представим себе базу данных, в которой хранится информация о протяжённых по времени событиях: для каждого события хранится промежуток времени, которое оно занимает. Рассматриваемая в этом разделе структура данных позволяет по любому промежутку найти все события, которые пересекаются с этим промежутком, причём делает это достаточно быстро.

Мы считаем, что отрезок $[t_1, t_2]$ представляет собой запись i, состоящую из двух полей: $low[i] = t_1$ (**левый конец** (low endpoint)) и $high[i] = t_2$ (**правый конец** (high endpoint)). Будем говорить, что отрезки i и i' **перекрываются** (overlap), если $low[i] \leqslant high[i']$ и $low[i'] \leqslant high[i]$; иными словами, если $i \cap i' \neq \emptyset$. (Обратите внимание, что отрезки, имеющие общий конец, считаются перекрывающимися.)

Всего возможно три варианта взаимного расположения отрезков i и i' (рис. 15.3):

1. отрезки i и i' перекрываются,
2. $high[i] < low[i']$,
3. $high[i'] < low[i]$.

Деревом промежутков (interval tree) назовём красно-чёрное дерево, каждая вершина x которого хранит отрезок $int[x]$. Дерево промежутков позволяет реализовать следующие операции:

INTERVAL-INSERT(T, x) добавляет к дереву T элемент x (содержащий некоторый отрезок $int[x]$);

INTERVAL-DELETE(T, x) удаляет из дерева T элемент x;

INTERVAL-SEARCH(T, i) возвращает указатель на элемент x дерева T, для которого отрезки i и $int[x]$ перекрываются (и возвращает NIL, если такого элемента в дереве нет).

Пример дерева промежутков показан на рис.15.4. Следуя схеме раздела 15.2, мы реализуем такую структуру данных и операции на

Рисунок 15.3 Три варианта взаимного расположения отрезков i и i'. (а) Отрезки i и i' перекрываются. Возможно четыре варианта; во всех четырёх $low[i] \leqslant high[i']$ и $low[i'] \leqslant high[i]$. (б) $high[i] < low[i']$. (в) $high[i'] < low[i]$.

ней.

Шаг 1: Базовая структура данных

Мы уже выбрали базовую структуру: красно-чёрное дерево, каждая вершина x которого содержит отрезок $int[x]$. Ключом вершины является левый конец отрезка $low[int[x]]$; обход дерева в порядке *"левое поддерево — корень — правое поддерево"* перечисляет вершины в порядке возрастания ключей.

Шаг 2: Дополнительная информация

Каждая вершина, помимо отрезка, содержит поле $max[x]$, в котором хранится максимальный из правых концов отрезков, содержащихся в поддереве с корнем x.

Шаг 3: Обновление дополнительной информации

Проверим, что дополнительную информацию можно обновлять при добавлении и удалении элемента без (асимптотического) ухудшения времени работы этих операций. В самом деле,

$$max[x] = \max(high[int[x]], max[left[x]], max[right[x]]),$$

и остаётся лишь сослаться на теорему 15.1. Можно отметить также, что при вращениях поле *max* можно обновлять за время $O(1)$ (упр. 15.2-4 и 15.3-1).

Шаг 4: Новые операции

Процедура Interval-Search(T, i) находит в дереве T отрезок, перекрывающийся с i. Если такого отрезка нет, она возвращает

Рисунок 15.4 Дерево промежутков. (а) Набор из 10 отрезков (выше тот, у которого левый конец больше). (б) Дерево промежутков, хранящее эти отрезки. При этом свойство упорядоченности дерева выполняется для левых концов.

значение NIL.

```
INTERVAL-SEARCH($T$, $i$)
$x \leftarrow root[T]$
while $x \neq$ NIL и $int[x]$ не перекрывается с $i$
    do if $left[x] \neq$ NIL и $max[left[x]] \geqslant low[i]$
          then $x \leftarrow left[x]$
          else $x \leftarrow right[x]$
return $x$
```

Мы ищем отрезок, проходя дерево от корня к листу. Процедура останавливается, если отрезок найден или если значение переменной x стало равным NIL. Каждая итерация цикла требует $O(1)$ шагов, поэтому время работы процедуры пропорционально высоте дерева (и равно $O(\log n)$ для дерева из n вершин).

Для примера посмотрим, как процедура ищет в дереве на рис. 15.4 отрезок, перекрывающийся с отрезком $i = [22, 25]$. Мы начинаем с корня ($x = root[T]$), который хранит отрезок $[16, 21]$, не перекрывающийся с i. Так как $max[left[x]] = 23$, что больше, чем

$low[i] = 22$, то мы переходим к левому ребёнку корня ($x \leftarrow left[x]$). Этот ребёнок хранит отрезок $[8, 9]$, также не перекрывающийся с i. На этот раз $max[left[x]] = 10$ меньше $low[i] = 22$, поэтому мы переходим к правому ребёнку вершины x. Там находится отрезок $[15, 23]$, перекрывающийся с i, и поиск завершается.

Рассмотрим пример безрезультатного поиска в том же дереве — будем искать отрезок, перекрывающийся с $i = [11, 14]$. Снова начинаем с корня. Корень хранит отрезок $[16, 21]$, не перекрывающийся с i. Так как $max[left[x]] = 23$ больше $low[i] = 11$, переходим к левому ребёнку. Теперь в x хранится отрезок $[8, 9]$. Он не перекрывается с i, и $max[left[x]] = 10$ меньше $low[i] = 11$, поэтому идём направо. (Слева искомого отрезка быть не может). В x теперь хранится $[15, 23]$, с i этот отрезок не перекрывается, $left[x] =$ NIL, поэтому идём направо и возвращаем значение NIL.

Корректность процедуры INTERVAL-SEARCH устанавливает теорема 15.2, которая утверждает, что если отрезки $int[x]$ и i не перекрываются, то дальнейший поиск идёт в правильном направлении (если нужные отрезки вообще есть в дереве, то они есть и в выбираемой части дерева). Поэтому нам достаточно просмотреть всего один путь. (Обратите внимание, что слова *"правильное направление"* не означают, что в другом направлении искомых отрезков нет: мы утверждаем лишь, что если они есть вообще, то есть и в *"правильном"* направлении!)

Теорема 15.2. *Пусть x — произвольная вершина дерева, i — отрезок, не перекрывающийся с $int[x]$, и мы выполняем строки 3–5 процедуры* INTERVAL-SEARCH(T, i). *Тогда:*

1. *Если выполняется строка 4, то либо поддерево с корнем $left[x]$ (левое поддерево) содержит отрезок, перекрывающийся с i, либо поддерево с корнем $right[x]$ (правое поддерево) не содержит отрезка, перекрывающегося с i.*
2. *Если выполняется строка 5, то левое поддерево не содержит отрезка, перекрывающегося с i.*

Доказательство. Начнём с более простого случая 2. Строка 5 выполняется, если не выполнено условие в строке 3, то есть если $left[x] =$ NIL или $max[left[x]] < low[i]$. В первом случае левое поддерево пусто, поэтому не содержит отрезка, перекрывающегося с i. Предположим, что $left[x] \neq$ NIL и $max[left[x]] < low[i]$. Рассмотрим произвольный отрезок i' из левого поддерева (см. рис. 15.5а). Так как $max[left[x]]$ — наибольший правый конец таких отрезков, то

$$high[i'] \leqslant max[left[i]] < low[i],$$

поэтому отрезок i' целиком лежит левее отрезка i. Случай 2 рассмотрен.

Рисунок 15.5 Отрезки в доказательстве теоремы 15.2. Пунктиром показано значение $max[left[x]]$. (а) Случай 2: мы идем направо. Никакой интервал i' не перекрывается с i. (б) Случай 1: мы идем налево. Выберем в левом поддереве отрезок i', для которого $high[i'] = max[left[x]] \geqslant low[i]$. Тогда либо i' перекрывается с i, либо i' лежит целиком справа от i, и i не перекрывается ни с каким отрезком i'' из правого поддерева, потому что $low[i'] \leqslant low[i'']$.

Рассмотрим теперь случай 1. Предположим, что в левом поддереве нет отрезков, перекрывающихся с i. Тогда нужно доказать, что таких отрезков нет и в правом. Так как выполняется строка 4, то условие в строке 3 выполнено, и $max[left[x]] \geqslant low[i]$. Поэтому в левом поддереве найдётся отрезок i', для которого

$$high[i'] = max[left[i']] \geqslant low[i]$$

(см. рис. 15.5б). Но отрезки i и i' не перекрываются (по предположению — мы считаем, что в левом поддереве нет отрезков, перекрывающихся с i). Это означает, что отрезок i' лежит целиком справа от i (целиком слева он лежать не может), то есть $high[i] < low[i']$. В дереве T выполнено свойство упорядоченности левых концов, поэтому и все отрезки правого поддерева лежат целиком справа от i: для произвольного отрезка i'' из правого поддерева выполнено

$$high[i] < low[i'] \leqslant low[i''].$$

□

Упражнения

15.3-1 Напишите процедуру Left-Rotate для дерева промежутков, которая обновляет поля max за время $O(1)$.

15.3-2 Перепишите процедуру Interval-Search для случая, когда в дереве промежутков хранятся не отрезки, а интервалы (то есть нас интересуют перекрытия ненулевой длины).

15.3-3 Постройте алгоритм, который по заданному отрезку возвращает перекрывающийся с ним отрезок с минимальным левым концом (или константу NIL, если таких отрезков нет).

15.3-4 Напишите программу, которая по заданному отрезку i и дереву промежутков T возвращает список всех отрезков дерева T, перекрывающихся с i. Время работы должно быть $O(\min(n, k\log n))$, где k — число элементов возвращаемого списка. (Дополнительный вопрос: как сделать это, не меняя дерева?)

15.3-5 Какие надо внести изменения в определение дерева промежутков, чтобы дополнительно реализовать процедуру INTERVAL-SEARCH-EXACTLY(T, i), которая либо возвращает вершину x дерева T, для которой $low[int[x]] = low[i]$ и $high[int[x]] = high[i]$, либо выдаёт константу NIL, если таких вершин нет (ищет отрезок, равный i, а не просто перекрывающийся с i.) Время работы процедуры INTERVAL-SEARCH-EXACTLY, а также всех прежних процедур промежутков должно оставаться равным $O(\log n)$.

15.3-6 Рассмотрим множество Q натуральных чисел. Определим MIN-GAP как расстояние между двумя ближайшими числами в Q. Например, если $Q = \{1, 5, 9, 15, 18, 22\}$, то MIN-GAP$(Q)$ равно $18 - 15 = 3$. Реализуйте структуру данных, эффективно реализующую добавление, удаление и поиск элемента, а также операцию MIN-GAP. Каково время работы этих операций?

15.3-7* При разработке интегральных схем информация часто представляется в виде списка прямоугольников. Пусть даны n прямоугольников со сторонами, параллельными осям координат. Каждый прямоугольник задаётся четырьмя числами: координатами левого нижнего и правого верхнего углов. Написать программу, которая за время $O(n\log n)$ выясняет, есть ли среди этих прямоугольников два перекрывающихся (но не требуется найти все перекрывающиеся прямоугольники). Обратите внимание, что границы перекрывающихся прямоугольников могут не пересекаться, если один лежит внутри другого. (Указание. Двигаем горизонтальную прямую снизу вверх и смотрим, как меняется её пересечение с прямоугольниками.)

Задачи

15-1 Точка максимальной кратности

Мы хотим следить за **точкой максимальной кратности** (point of maximum overlap) множества промежутков — точкой, которая принадлежит максимальному числу промежутков из этого множества. Как нам обновлять информацию об этой точке при добавлении и удалении элемента?

15-2 Детская считалка

Задача о детской считалке (Josephus problem) состоит в следующем. Фиксируем два числа m и n ($m \leqslant n$). Вначале n детей стоят по кругу. Начав с кого-то, мы считаем *"первый, второй, третий..."*Как только доходим до m-го, он выходит из круга, и счёт продолжается дальше по кругу уже без него (начиная с единицы). Так продолжается, пока не останется ровно один человек. Нас будет интересовать зависящая от n и m последовательность ((n,m)-Josephus permutation), в которой дети выходят из круга. Например, при $n = 7$ и $m = 3$ искомая последовательность выглядит так: $\langle 3,6,2,7,5,1,4 \rangle$.

а Зафиксируем m. Напишите программу, которая по данному n даёт эту последовательность за время $O(n)$.

б Напишите программу, которая по произвольным n и m даёт искомую последовательность за время $O(n \log n)$.

Замечания

Описания некоторых видов деревьев промежутков можно найти в Препарата и Шамос [160]. Наибольший теоретический интерес представляет здесь результат, к которому независимо пришли Эдельсбруннер (H. Edelsbrunner, 1980) и МакКрейт (E. M. McCreight, 1981). Разработанная ими структура данных хранит n отрезков и позволяет перечислить те из них, которые перекрываются с заданным отрезком, за время $O(k + \log n)$, где k — количество таких отрезков.

IV Методы построения и анализа алгоритмов

Введение

Эта часть посвящена трём важным методам построения и анализа эффективных алгоритмов: динамическому программированию (глава 16), жадным алгоритмам (глава 17) и амортизационному анализу (глава 18). Эти методы, возможно, чуть сложнее разобранных ранее (*"разделяй и властвуй"*, использование случайных чисел, решение рекуррентных соотношений), но не менее важны.

Динамическое программирование обычно применяется к задачам, в которых искомый ответ состоит из частей, каждая из которых в свою очередь даёт оптимальное решение некоторой подзадачи. Динамическое программирование полезно, если на разных путях многократно встречаются одни и те же подзадачи; основной технический приём — запоминать решения встречающихся подзадач на случай, если та же подзадача встретится вновь.

Жадные алгоритмы, как и динамическое программирование, применяются в тех случаях, когда искомый объект строится по частям. Жадный алгоритм делает на каждом шаге *"локально оптимальный"* выбор. Простой пример: стараясь набрать данную сумму денег минимальным числом монет, можно последовательно брать монеты наибольшего возможного достоинства (не превосходящего той суммы, которую осталось набрать).

Жадный алгоритм обычно работает гораздо быстрее, чем алгоритм, основанный на динамическом программировании. Однако жадный алгоритм вовсе не всегда даёт оптимальное решение. Во многих задачах применимость жадных алгоритмов удаётся доказать с помощью так называемых матроидов, о которых рассказано в главе 17.

Амортизационный анализ — это средство анализа алгоритмов, производящих последовательность однотипных операций. Вместо того, чтобы оценивать время работы для каждой из этих операций по отдельности, амортизационный анализ оценивает среднее время работы в расчёте на одну операцию. Разница может оказаться существенной, если долго выполняемые операции не могут идти подряд. На самом деле амортизационный анализ — не только средство анализа алгоритмов, но еще и подход к разработке алгоритмов: ведь разработка алгоритма и анализ скорости его работы тесно связаны. В главе 18 мы рассказываем о трёх методах амортизационного анализа алгоритмов.

16 Динамическое программирование

Подобно методу *"разделяй и властвуй"*, динамическое программирование решает задачу, разбивая её на подзадачи и объединяя их решения. Как мы видели в главе 1, алгоритмы типа *"разделяй и властвуй"* делят задачу на независимые подзадачи, эти подзадачи — на более мелкие подзадачи и так далее, а затем собирают решение основной задачи *"снизу вверх"*. Динамическое программирование применимо тогда, когда подзадачи не являются независимыми, иными словами, когда у подзадач есть общие *"подподзадачи"*. В этом случае алгоритм типа *"разделяй и властвуй"* будет делать лишнюю работу, решая одни и те же подподзадачи по нескольку раз. Алгоритм, основанный на динамическом программировании, решает каждую из подзадач единожды и запоминает ответы в специальной таблице. Это позволяет не вычислять заново ответ к уже встречавшейся подзадаче.

В типичном случае динамическое программирование применяется к **задачам оптимизации** (optimization problems). У такой задачи может быть много возможных решений; их *"качество"* определяется значением какого-то параметра, и требуется выбрать оптимальное решение, при котором значение параметра будет минимальным или максимальным (в зависимости от постановки задачи). Вообще говоря, оптимум может достигаться для нескольких разных решений.

Как строится алгоритм, основанный на динамическом программировании? Надо:

1. описать структуру оптимальных решений,
2. выписать рекуррентное соотношение, связывающее оптимальные значения параметра для подзадач,
3. двигаясь снизу вверх, вычислить оптимальное значение параметра,
4. пользуясь полученной информацией, построить оптимальное решение.

Основную часть работы составляют шаги 1–3. Если нас интересует только оптимальное значение параметра, шаг 4 не нужен. Если же шаг 4 необходим, для построения оптимального решения иногда приходится получать и хранить дополнительную информацию в процессе выполнения шага 3.

В этой главе мы решим несколько оптимизационных задач с помощью динамического программирования. В разделе 16.1 мы выясним, как найти произведение нескольких матриц, сделав как можно меньше умножений. В разделе 16.2 мы обсудим, какие свойства задачи делают возможным применение динамического программирования. После этого мы расскажем (в разделе 16.3), как найти наибольшую общую подпоследовательность двух последовательностей. Наконец, в разделе 16.4 мы воспользуемся динамическим программированием для нахождения оптимальной триангуляции выпуклого многоугольника. (Удивительным образом эта задача оказывается похожей на задачу о перемножении нескольких матриц.)

16.1 Перемножение нескольких матриц

Мы хотим найти произведение

$$A_1 A_2 \ldots A_n \tag{16.1}$$

последовательности n матриц $\langle A_1, A_2, \ldots, A_n \rangle$. Мы будем пользоваться стандартным алгоритмом перемножения двух матриц в качестве подпрограммы. Но прежде надо расставить скобки в (16.1), чтобы указать порядок умножений. Будем говорить, что в произведении матриц **полностью расставлены скобки** (product is fully parenthesized), если это произведение либо состоит из одной-единственной матрицы, либо является заключенным в скобки произведением двух произведений с полностью расставленными скобками. Поскольку умножение матриц ассоциативно, конечный результат вычислений не зависит от расстановки скобок. Например, в произведении $A_1 A_2 A_3 A_4$ можно полностью расставить скобки пятью разными способами:

$$(A_1(A_2(A_3A_4))); \quad (A_1((A_2A_3)A_4)); \quad ((A_1A_2)(A_3A_4));$$
$$((A_1(A_2A_3))A_4); \quad (((A_1A_2)A_3)A_4);$$

во всех случаях ответ будет один и тот же.

Не влияя на ответ, способ расстановки скобок может сильно повлиять на стоимость перемножения матриц. Посмотрим сначала, сколько операций требует перемножение двух матриц. Вот стандартный алгоритм (*rows* и *columns* обозначают количество строк и столбцов матрицы соответственно):

```
MATRIX-MULTIPLY$(A, B)$
1  if $columns[A] \neq rows[B]$
2     then error "умножить нельзя"
3     else for $i \leftarrow 1$ to $rows[A]$
4              do for $j \leftarrow 1$ to $columns[B]$
5                     do $C[i,j] \leftarrow 0$
6                        for $k \leftarrow 1$ to $columns[A]$
7                            do $C[i,j] \leftarrow C[i,j] + A[i,k] \cdot B[k,j]$
8           return $C$
```

Матрицы A и B можно перемножать, только если число столбцов у A равно числу строк у B. Если A — это $p \times q$-матрица, а B — это $q \times r$-матрица, то их произведение C является $p \times r$-матрицей. При выполнении этого алгоритма делается pqr умножений (строка 7) и примерно столько же сложений. Для простоты мы будем учитывать только умножения. [В главе 31 приведён алгоритм Штрассена, требующий меньшего числа умножений. В этой главе мы принимаем как данность простейший способ умножения матриц и ищем оптимум за счёт расстановки скобок.]

Чтобы увидеть, как расстановка скобок может влиять на стоимость, рассмотрим последовательность из трех матриц $\langle A_1, A_2, A_3 \rangle$ размеров 10×100, 100×5 и 5×50 соответственно. При вычислении $((A_1A_2)A_3)$ нужно $10 \cdot 100 \cdot 5 = 5000$ умножений, чтобы найти 10×5-матрицу A_1A_2, а затем $10 \cdot 5 \cdot 50 = 2500$ умножений, чтобы умножить эту матрицу на A_3. Всего 7500 умножений. При расстановке скобок $(A_1(A_2A_3))$ мы делаем $100 \times 5 \times 50 = 25\,000$ умножений для нахождения 100×50-матрицы A_2A_3, плюс ещё $10 \times 100 \times 50 = 50\,000$ умножений (умножение A_1 на A_2A_3), итого 75 000 умножений. Тем самым, первый способ расстановки скобок в 10 раз выгоднее.

Задача об умножении последовательности матриц (matrix-chain multiplication problem) может быть сформулирована следующим образом: дана последовательность из n матриц $\langle A_1, A_2, \ldots, A_n \rangle$ заданных размеров (матрица A_i имеет размер $p_{i-1} \times p_i$); требуется найти такую (полную) расстановку скобок в произведении $A_1A_2 \ldots A_n$, чтобы вычисление произведения требовало наименьшего числа умножений.

Количество расстановок скобок

Прежде чем применять динамическое программирование к задаче об умножении последовательности матриц, стоит убедиться, что простой перебор всех возможных расстановок скобок не даст эффективного алгоритма. Обозначим символом $P(n)$ количество полных расстановок скобок в произведении n матриц. Последнее умножение может происходить на границе между k-й и $(k+1)$-й матрицами. До этого мы отдельно вычисляем произведение первых k и остальных $n-k$ матриц. Поэтому

$$P(n) = \begin{cases} 1, & \text{если } n = 1, \\ \sum\limits_{k=1}^{n-1} P(k)P(n-k), & \text{если } n \geqslant 2. \end{cases}$$

В задаче 13-4 мы просили вас доказать, что это соотношение задаёт последовательность **чисел Каталана**

$$P(n) = C(n-1),$$

где

$$C(n) = \frac{1}{n+1} C_{2n}^{n} = \Omega(4^n / n^{3/2}).$$

Стало быть, число решений экспоненциально зависит от n, так что полный перебор неэффективен. [Другой способ понять, что число вариантов экспоненциально: разобьём матрицы на группы по три; произведение для каждой группы можно вычислить двумя способами, так что для $3n$ матриц есть не менее 2^n вариантов.]

Шаг 1: строение оптимальной расстановки скобок

Если мы собираемся воспользоваться динамическим программированием, то для начала должны описать строение оптимальных решений. Для задачи об умножении последовательности матриц это выглядит следующим образом. Обозначим для удобства через $A_{i..j}$ матрицу, являющуюся произведением $A_i A_{i+1} \ldots A_j$. Оптимальная расстановка скобок в произведении $A_1 A_2 \ldots A_n$ разрывает последовательность между A_k и A_{k+1} для некоторого k, удовлетворяющего неравенству $1 \leqslant k < n$. Иными словами, при вычислении произведения, диктуемом этой расстановкой скобок, мы сначала вычисляем произведения $A_{1..k}$ и $A_{k+1..n}$, а затем перемножаем их и получаем окончательный ответ $A_{1..n}$. Стало быть, стоимость этой оптимальной расстановки равна стоимости вычисления матрицы $A_{1..k}$, плюс стоимость вычисления матрицы $A_{k+1..n}$, плюс стоимость перемножения этих двух матриц.

Чем меньше умножений нам потребуется для вычисления $A_{1..k}$ и $A_{k+1..n}$, тем меньше будет общее число умножений. Стало быть, оптимальное решение задачи о перемножении последовательности матриц содержит оптимальные решения задач о перемножении её частей. Как мы увидим в разделе 16.2, это и позволяет применить динамическое программирование.

Шаг 2: рекуррентное соотношение

Теперь надо выразить стоимость оптимального решения задачи через стоимости оптимальных решений её подзадач. Такими подзадачами будут задачи об оптимальной расстановке скобок в произведениях $A_{i..j} = A_i A_{i+1} \dots A_j$ для $1 \leqslant i \leqslant j \leqslant n$. Обозначим через $m[i,j]$ минимальное количество умножений, необходимое для вычисления матрицы $A_{i..j}$; в частности, стоимость вычисления всего произведения $A_{1..n}$ есть $m[1,n]$.

Числа $m[i,j]$ можно вычислить так. Если $i = j$, то последовательность состоит из одной матрицы $A_{i..i} = A_i$ и умножения вообще не нужны. Стало быть, $m[i,i] = 0$ для $i = 1, 2, \dots, n$. Чтобы посчитать $m[i,j]$ для $i < j$, мы воспользуемся информацией о строении оптимального решения, полученной на шаге 1. Пусть при оптимальной расстановке скобок в произведении $A_i A_{i+1} \dots A_j$ последним идет умножение $A_i \dots A_k$ на $A_{k+1} \dots A_j$, где $i \leqslant k < j$. Тогда $m[i,j]$ равно сумме минимальных стоимостей вычисления произведений $A_{i..k}$ и $A_{k+1..j}$ плюс стоимость перемножения этих двух матриц. Поскольку для вычисления произведения $A_{i..k} A_{k+1..j}$ требуется $p_{i-1} p_k p_j$ умножений,

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j.$$

В этом соотношении подразумевается, что оптимальное значение k нам известно; на деле это не так. Однако число k может принимать всего лишь $j - i$ различных значений: $i, i+1, \dots, j-1$. Поскольку одно из них оптимально, достаточно перебрать эти значения k и выбрать наилучшее. Получаем рекуррентную формулу:

$$m[i,j] = \begin{cases} 0 & \text{при } i = j, \\ \min\limits_{i \leqslant k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{при } i < j. \end{cases} \tag{16.2}$$

Числа $m[i,j]$ — стоимости оптимальных решений подзадач. Чтобы проследить за тем, как получается оптимальное решение, обозначим через $s[i,j]$ оптимальное место последнего умножения, то есть такое k, что при оптимальном вычислении произведения $A_i A_{i+1} \dots A_j$ последним идет умножение $A_i \dots A_k$ на $A_{k+1} \dots A_j$. Иными словами, $s[i,j]$ равно числу k, для которого $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$.

Шаг 3: вычисление оптимальной стоимости

Пользуясь соотношениями (16.2), теперь легко написать рекурсивный алгоритм, определяющий минимальную стоимость вычисления произведения $A_1 A_2 \dots A_n$ (т.е. число $m[1,n]$). Однако время работы такого алгоритма экспоненциально зависит от n, так что этот алгоритм не лучше полного перебора. Настоящий выигрыш во времени мы получим, если воспользуемся тем, что подзадач относительно немного: по одной задаче для каждой пары (i, j), для которой $1 \leqslant i \leqslant j \leqslant n$, а всего $C_n^2 + n = \Theta(n^2)$. Экспоненциальное время работы возникает потому, что рекурсивный алгоритм решает каждую из подзадач по многу раз, на разных ветвях дерева рекурсии. Такое *"перекрытие подзадач"* — характерный признак задач, решаемых методом динамического программирования.

Вместо рекурсии мы вычислим оптимальную стоимость *"снизу вверх"*. В нижеследующей программе предполагается, что матрица A_i имеет размер $p_{i-1} \times p_i$ при $i = 1, 2, \ldots, n$. На вход подаётся последовательность $p = \langle p_0, p_1, \ldots, p_n \rangle$, где $length[p] = n + 1$. Программа использует вспомогательные таблицы $m[1\,..\,n, 1\,..\,n]$ (для хранения стоимостей $m[i, j]$) и $s[1\,..\,n, 1\,..\,n]$ (в ней отмечается, при каком k достигается оптимальная стоимость при вычислении $m[i, j]$).

```
MATRIX-CHAIN-ORDER($p$)
 1  $n \leftarrow length[p] - 1$
 2  for $i \leftarrow 1$ to $n$
 3      do $m[i, i] \leftarrow 0$
 4  for $l \leftarrow 2$ to $n$
 5      do for $i \leftarrow 1$ to $n - l + 1$
 6             do $j \leftarrow i + l - 1$
 7                $m[i, j] \leftarrow \infty$
 8                for $k \leftarrow i$ to $j - 1$
 9                    do $q \leftarrow m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$
10                       if $q < m[i, j]$
11                         then $m[i, j] \leftarrow q$
12                              $s[i, j] \leftarrow k$
13  return $m$, $s$
```

Заполняя таблицу m, этот алгоритм последовательно решает задачи об оптимальной расстановке скобок для одного, двух, …, n сомножителей. В самом деле, соотношение (16.2) показывает, что число $m[i, j]$ — стоимость перемножения $j - i + 1$ матриц — зависит только от стоимостей меньшего (чем $j - i + 1$) числа матриц. Именно, для $k = i, i + 1, \ldots, j - 1$ получается, что $A_{i..k}$ — произведение $k - i + 1 < j - i + 1$ матриц, а $A_{k+1..j}$ — произведение $j - k < j - i + 1$ матриц.

Сначала (в строках 2–3) алгоритм выполняет присваивания $m[i, i] \leftarrow 0$ для $i = 1, 2, \ldots, n$: стоимость перемножения последовательности из одной матрицы равна нулю. При первом исполнении цикла (строки 4–12) вычисляются (с помощью соотношений (16.2)) значения $m[i, i + 1]$ для $i = 1, 2, \ldots, n - 1$ — это минимальные стоимости для последовательностей длины 2. При втором проходе вычи-

Рисунок 16.1 Таблицы m и s, вычисляемые процедурой Matrix-Chain-Order для $n = 6$ и матриц следующего размера:

матрица	размер
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

Таблицы повёрнуты так, что главная диагональ горизонтальна. В таблице m используются только клетки, лежащие не ниже главной диагонали, в таблице s — только клетки, лежащие строго выше. Минимальное количество умножений, необходимое для перемножения всех шести матриц, равно $m[1,6] = 15\,125$. Пары клеточек, заштрихованных одинаковой светлой штриховкой, совместно входят в правую часть формулы в процессе вычисления $m[2,5]$ (строка 9 процедуры Matrix-Chain-Order):

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375. \end{cases}$$

сляются $m[i, i+2]$ для $i = 1, 2, \ldots, n-2$ — минимальные стоимости перемножения последовательностей длины 3, и так далее. На каждом шаге значение $m[i,j]$, вычисляемое в строках 9–12, зависит только от вычисленных ранее значений $m[i,k]$ и $m[k+1,j]$.

На рис. 16.1 показано, как это происходит при $n = 6$. Поскольку мы определяем $m[i,j]$ только для $i \leqslant j$, используется часть таблицы, лежащая над главной диагональю. На рисунке таблицы повёрнуты (главная диагональ горизонтальна). Внизу выписана последовательность матриц. Число $m[i,j]$ — минимальная стоимость перемножения подпоследовательности $A_i A_{i+1} \ldots A_j$ — находится на пересечении диагоналей, идущих вправо-вверх от матрицы A_i и влево-вверх от матрицы A_j. В каждом горизонтальном ряду собраны стоимости перемножения подпоследовательностей фиксированной длины. Для заполнения клетки $m[i,j]$ нужно знать произведения $p_{i-1} p_k p_j$ для $k = i, i+1, \ldots, j-1$ и содержимое клеток, лежащих слева-внизу и справа-внизу от $m[i,j]$.

Простая оценка показывает, что время работы алгоритма Matrix-Chain-Order есть $O(n^3)$. В самом деле, число вложенных циклов равно трём, и каждый из индексов l, i и k принимает не более n значений. В упражнении 16.1-3 мы предложим вам показать, что время работы этого алгоритма есть $\Theta(n^3)$. Объём памяти, необходимый для хранения таблиц m и s, есть $\Theta(n^2)$. Тем самым этот алгоритм значительно эффективнее, чем требующий экспоненциального времени перебор всех расстановок.

Шаг 4: построение оптимального решения

Алгоритм Matrix-Chain-Order находит минимальное число умножений, необходимое для перемножения последовательности матриц. Осталось найти расстановку скобок, приводящую к такому числу умножений.

Для этого мы используем таблицу $s[1\,..\,n, 1\,..\,n]$. В клетке $s[i,j]$ записано место последнего умножения при оптимальной расстановке скобок; другими словами, при оптимальном способе вычисления $A_{1..n}$ последним идёт умножение $A_{1..s[1,n]}$ на $A_{s[1,n]+1..n}$. Предшествующие умножения можно найти рекурсивно: значение $s[1, s[1,n]]$ определяет последнее умножение при нахождении $A_{1..s[1,n]}$, а $s[s[1,n]+1, n]$ определяет последнее умножение при вычислении $A_{s[1,n]+1..n}$. Приведённая ниже рекурсивная процедура вычисляет произведение $A_{i..j}$, имея следующие данные: последовательность матриц $A = \langle A_1, A_2, \ldots, A_n \rangle$, таблицу s, найденную процедурой Matrix-Chain-Order, а также индексы i и j. Произведение $A_1 A_2 \ldots A_n$ равно Matrix-Chain-Multiply$(A, s, 1, n)$.

```
Matrix-Chain-Multiply$(A, s, i, j)$
if $j > i$
   then $X \leftarrow$ Matrix-Chain-Multiply$(A, s, i, s[i,j])$
        $Y \leftarrow$ Matrix-Chain-Multiply$(A, s, s[i,j]+1, j)$
        return Matrix-Multiply$(X, Y)$
   else return $A_i$
```

В примере на рис. 16.1 вызов Matrix-Chain-Multiply$(A, s, 1, 6)$ вычислит произведение шести матриц в соответствии с расстановкой скобок

$$((A_1(A_2A_3))((A_4A_5)A_6)). \tag{16.3}$$

[Техническое замечание: следует позаботиться, чтобы при передаче массива s в процедуру не происходило копирования.]

Упражнения

16.1-1 Найдите оптимальную расстановку в задаче о перемножении матриц, если $p = \langle 5, 10, 3, 12, 5, 50, 6 \rangle$.

16.1-2 Разработайте алгоритм Print-Optimal-Parens, печатающий оптимальную расстановку скобок. (Таблица s уже вычислена алгоритмом Matrix-Chain-Order.)

16.1-3 Пусть $R(i, j)$ обозначает количество обращений алгоритма Matrix-Chain-Order к элементу $m[i, j]$ таблицы m с целью вычисления других элементов таблицы [строка 9]. Покажите, что общее количество таких обращений равно

$$\sum_{i=1}^{n} \sum_{j=1}^{n} R(i, j) = \frac{n^3 - n}{3}.$$

(Указание. Вам может пригодиться формула $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$.)

16.1-4 Покажите, что полная расстановка скобок в произведении n множителей использует ровно $n-1$ пар скобок.

16.2 Когда применимо динамическое программирование

В этом разделе мы укажем два признака, характерных для задач, решаемых методом динамического программирования.

Оптимальность для подзадач

При решении оптимизационной задачи с помощью динамического программирования необходимо сначала описать структуру оптимального решения. Будем говорить, что задача **обладает свойством оптимальности для задач** (has optimal substructure), если оптимальное решение задачи содержит оптимальные решения её подзадач. Если задача обладает этим свойством, то динамическое программирование может оказаться полезным для её решения (а возможно, применим и жадный алгоритм — см. главу 17).

В разделе 16.1 мы видели, что задача перемножения матриц обладает свойством оптимальности для подзадач: каждая скобка в оптимальном произведении указывает оптимальный способ перемножения входящих в неё матриц. Чтобы убедиться, что задача обладает этим свойством, надо (как в разделе 16.1) показать, что, улучшая решение подзадачи, мы улучшим и решение исходной задачи.

Как только свойство оптимальности для подзадач установлено, обычно становится ясно, с каким именно множеством подзадач будет иметь дело алгоритм. Например, для задачи о перемножении последовательности матриц подзадачами будут задачи о перемножении кусков этой последовательности.

Перекрывающиеся подзадачи

Второй свойство задач, необходимое для использования динамического программирования, — малость множества подзадач. Благодаря этому при рекурсивном решении задачи мы всё время выходим на одни и те же подзадачи. В таком случае говорят, что у оптимизационной задачи имеются **перекрывающиеся подзадачи** (overlapping subproblems). В типичных случаях количество подзадач полиномиально зависит от размера исходных данных.

В задачах, решаемых методом *"разделяй и властвуй"*, так не бывает: для них рекурсивный алгоритм, как правило, на каждом шаге порождает совершенно новые подзадачи. Алгоритмы, основанные на динамическом программировании, используют перекрытие подзадач следующим образом: каждая из подзадач решается только один раз, и ответ заносится в специальную таблицу; когда эта же подзадача встречается снова, программа не тратит время на её решение, а берёт готовый ответ из таблицы.

Вернёмся для примера к задаче о перемножении последовательности матриц. Из рисунка 16.1 видно, что решение каждой из подзадач, записанное в данной клеточке таблицы, многократно используется процедурой MATRIX-CHAIN-ORDER при решении подзадач из расположенных выше клеточек. Например, $m[3,4]$ используется четырежды: при вычислении $m[2,4]$, $m[1,4]$, $m[3,5]$ и $m[3,6]$. Было бы крайне неэффективно вычислять $m[3,4]$ всякий раз заново. В самом деле, рассмотрим следующий (неэффективный) рекурсивный алгоритм, основанный непосредственно на соотношениях (16.2) и вычисляющий $m[i,j]$ — минимальное количество умножений, необходимое для вычисления $A_{i..j} = A_i A_{i+1} \ldots A_j$:

```
RECURSIVE-MATRIX-CHAIN($p, i, j$)
if $i = j$
   then return 0
$m[i,j] \leftarrow \infty$
for $k \leftarrow i$ to $j - 1$
     do $q \leftarrow$ RECURSIVE-MATRIX-CHAIN($p, i, k$) +
              + RECURSIVE-MATRIX-CHAIN($p, k+1, j$) + $p_{i-1}p_kp_j$
        if $q < m[i,j]$
          then $m[i,j] \leftarrow q$
return $m[i,j]$
```

Рисунок 16.2 Дерево рекурсии для Recursive-Matrix-Chain$(p, 1, 4)$. В каждой вершине записаны значения i и j. Заштрихованы *"лишние"* вершины (вычисления в которых повторяют уже проделанные).

На рис. 16.2 изображено дерево рекурсии для Recursive-Matrix-Chain$(p, 1, 4)$. В каждой вершине записаны значения i и j. Обратите внимание, что некоторые пары (i, j) встречаются многократно.

Легко видеть, что время работы Recursive-Matrix-Chain$(p, 1, n)$ зависит от n по меньшей мере экспоненциально. В самом деле, обозначим его $T(n)$ и примем, что время исполнения строк 1–2, а также 6–7, равно единице. Тогда:

$$\begin{aligned} &T(1) \geqslant 1, \\ &T(n) \geqslant 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1), \qquad \text{если } n > 1. \end{aligned}$$

В сумме по k каждое $T(i)$ (при $i = 1, 2, \ldots, n-1$) встречается дважды, и ещё есть $n - 1$ единиц. Стало быть,

$$T(n) \geqslant 2 \sum_{i=1}^{n-1} T(i) + n. \tag{16.4}$$

Докажем по индукции, что $T(n) \geqslant 2^{n-1}$ для всех $n \geqslant 1$. При $n = 1$ неравенство выполнено, так как $T(1) \geqslant 1 = 2^0$. Шаг индукции:

$$\begin{aligned} T(n) &\geqslant 2 \sum_{i=1}^{n-1} 2^{i-1} + n = 2 \sum_{i=0}^{n-2} 2^i + n = \\ &= 2(2^{n-1} - 1) + n = 2^n - 2 + n \geqslant 2^{n-1}. \end{aligned}$$

Мы видим, что алгоритм Recursive-Matrix-Chain требует экспоненциального времени. Причина в том, что этот алгоритм многократно встречает одинаковые подзадачи и каждый раз решает их заново. Различных подзадач всего лишь $\Theta(n^2)$, и масса времени теряется на лишнюю работу. Метод динамического программирования позволяет этой лишней работы избежать.

Динамическое программирование *”сверху вниз”*

Алгоритм раздела 16.1 действовал *”снизу вверх”*. Но тот же приём (исключение повторного решения подзадач) можно реализовать и для алгоритмов, работающих *”сверху вниз”*. Для этого нужно запоминать ответы к уже решённым подзадачам в специальной таблице. Сначала вся таблица пуста (т.е. заполнена специальными записями, указывающими на то, что соответствующее значение еще не вычислено). Когда в процессе выполнения алгоритма подзадача встречается в первый раз, её решение заносится в таблицу. В дальнейшем решение этой подзадачи берётся прямо из таблицы. (В нашем примере таблицу ответов завести легко, так как подзадачи нумеруются парами (i,j). В более сложных случаях можно использовать хеширование.) По-английски этот прием улучшения рекурсивных алгоритмов называется **memoization**.

Применим это усовершенствование к алгоритму Recursive-Matrix-Chain:

```
Memoized-Matrix-Chain($p$)
  $n \leftarrow length[p] - 1$
  for $i \leftarrow 1$ to $n$
      do for $j \leftarrow i$ to $n$
             do $m[i,j] \leftarrow \infty$
  return Lookup-Chain($p, 1, n$)
```

```
Lookup-Chain($p, i, j$)
  if $m[i,j] < \infty$
     then return $m[i,j]$
  if $i = j$
     then $m[i,j] \leftarrow 0$
     else for $k \leftarrow i$ to $j - 1$
              do $q \leftarrow$ Lookup-Chain($p, i, k$) +
                        + Lookup-Chain($p, k+1, j$) + $p_{i-1}p_kp_j$
                 if $q < m[i,j]$
                    then $m[i,j] \leftarrow q$
  return $m[i,j]$
```

Процедура Memoized-Matrix-Chain, подобно Matrix-Chain-Order, заполняет таблицу $m[1\,..\,n, 1\,..\,n]$, где $m[i,j]$ — минимальное количество умножений, необходимое для вычисления $A_{i..j}$. Первоначально $m[i,j] = \infty$ в знак того, что соответствующее место в таблице не заполнено. Если при исполнении Lookup-Chain(p,i,j) оказывается, что $m[i,j] < \infty$, то процедура сразу выдает это значение $m[i,j]$. В противном случае $m[i,j]$ вычисляется как в процедуре Recursive-Matrix-Chain, записывается в таблицу и выдаётся в качестве ответа. Тем самым вызов Lookup-Chain(p,i,j) всегда возвращает $m[i, j]$, но вычисления проводятся только при первом таком вызове.

Рис. 16.2 иллюстрирует экономию, достигаемую заменой Recursive-Matrix-Chain на Memoized-Matrix-Chain. Заштрихованные вершины дерева рекурсии соответствуют тем случаям, когда значение $m[i, j]$ не вычисляется, а берётся прямо из таблицы.

Алгоритм Memoized-Matrix-Chain требует времени $O(n^3)$, как и алгоритм Matrix-Chain-Order. В самом деле, каждая из $\Theta(n^2)$ позиций таблицы один раз инициализируется (строка 4 процедуры Memoized-Matrix-Chain) и один-единственный раз заполняется — при первом вызове Lookup-Chain(p, i, j) для данных i и j. Все вызовы Lookup-Chain(p, i, j) делятся на первые и повторные. Каждый из $\Theta(n^2)$ первых вызовов требует времени $O(n)$ (не включая времени работы рекурсивных вызовов Lookup-Chain для меньших участков); общее время работы есть $O(n^3)$. Каждый из повторных вызовов требует времени $O(1)$; их число есть $O(n^3)$ (вычисления для каждой из $O(n^2)$ клеток таблицы порождают $O(n)$ вызовов). Тем самым рекурсивный алгоритм, требующий времени $\Omega(2^n)$, превратился в полиномиальный, требующий времени $O(n^3)$.

Подведём итоги: задача об оптимальном порядке умножения n матриц может быть решена за время $O(n^3)$ либо *"сверху вниз"* (рекурсивный алгоритм с запоминанием ответов), либо *"снизу вверх"* (динамическое программирование). Оба алгоритма основаны на перекрытии подзадач; число подзадач есть $\Theta(n^2)$, и оба алгоритма решают каждую из подзадач лишь единожды.

Вообще говоря, если каждая из подзадач должна быть решена хоть раз, метод динамического программирования (*"снизу вверх"*) обычно эффективнее, чем рекурсия с запоминанием ответов, поскольку реализация рекурсии (а также проверка, есть ответ в таблице или ещё нет) требует дополнительного времени. Но если для нахождения оптимума не обязательно решать все подзадачи, подход *"сверху вниз"* имеет то преимущество, что решаются лишь те подзадачи, которые действительно нужны.

Упражнения

16.2-1 Сравните неравенство (16.4) с формулой (8.4), использованной при оценке времени работы алгоритма быстрой сортировки. В чём причина того, что оценки, получающиеся из этих двух рекуррентных соотношений, столь различны?

16.2-2 Как лучше искать оптимальный порядок перемножения матриц: перебирая все расстановки скобок и вычисляя количество умножений для каждой из них, или же с помощью алгоритма

Recursive-Matrix-Chain?

16.2-3 Нарисуйте дерево рекурсии для алгоритма Merge-Sort (сортировка слиянием) из раздела 1.3.1, применённого к массиву из 16 элементов. Почему здесь нет смысла запоминать ответы к уже решённым подзадачам?

16.3 Наибольшая общая подпоследовательность

Подпоследовательность получается из данной последовательности, если удалить некоторые её элементы (сама последовательность также считается своей подпоследовательностью). Формально: последовательность $Z = \langle z_1, z_2, \ldots, z_k \rangle$ называется **подпоследовательностью** (subsequence) последовательности $X = \langle x_1, x_2, \ldots, x_n \rangle$, если существует строго возрастающая последовательность индексов $\langle i_1, i_2, \ldots, i_k \rangle$, для которой $z_j = x_{i_j}$ при всех $j = 1, 2, \ldots, k$. Например, $Z = \langle B, C, D, B \rangle$ является подпоследовательностью последовательности $X = \langle A, B, C, B, D, A, B \rangle$; соответствующая последовательность индексов есть $\langle 2, 3, 5, 7 \rangle$. (Отметим, что говоря о последовательностях, мы — в отличие от курсов математического анализа — имеем в виду конечные последовательности.)

Будем говорить, что последовательность Z является **общей подпоследовательностью** (common subsequence) последовательностей X и Y, если Z является подпоследовательностью как X, так и Y. Пример: $X = \langle A, B, C, B, D, A, B \rangle$, $Y = \langle B, D, C, A, B, A \rangle$, $Z = \langle B, C, A \rangle$. Последовательность Z в этом примере — не самая длинная из общих подпоследовательностей X и Y (последовательность $\langle B, C, B, A \rangle$ длиннее). Последовательность $\langle B, C, B, A \rangle$ будет наибольшей общей подпоследовательностью для X и Y, поскольку общих подпоследовательностей длины 5 у них нет. Наибольших общих подпоследовательностей может быть несколько. Например, $\langle B, D, A, B \rangle$ — другая наибольшая общая подпоследовательность X и Y.

Задача о наибольшей общей подпоследовательности (сокращенно НОП; по-английски LCS = longest-common-subsequence) состоит в том, чтобы найти общую подпоследовательность наибольшей длины для двух данных последовательностей X и Y. В этом разделе мы покажем, как решить эту задачу с помощью динамического программирования.

Строение наибольшей общей подпоследовательности

Если решать задачу о НОП *"в лоб"*, перебирая все подпоследовательности последовательности X и проверяя для каждой из них, не будет ли она подпоследовательностью последовательности Y, то алгоритм будет работать экспоненциальное время, поскольку последовательность длины m имеет 2^m подпоследовательностей (столько же, сколько подмножеств у множества $\{1, 2, \ldots, m\}$).

Однако задача о НОП обладает свойством оптимальности для подзадач, как показывает теорема 16.1 (см. ниже). Подходящее множество подзадач — множество пар префиксов двух данных последовательностей. Пусть $X = \langle x_1, x_2, \ldots, x_m \rangle$ — некоторая последовательность. Её **префикс** (prefix) длины i — это последовательность $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ (при i от 0 до m). Например, если $X = \langle A, B, C, B, D, A, B \rangle$, то $X_4 = \langle A, B, C, B \rangle$, а X_0 — пустая последовательность.

Теорема 16.1 (о строении НОП). *Пусть* $Z = \langle z_1, z_2, \ldots, z_k \rangle$ — *одна из наибольших общих подпоследовательностей для* $X = \langle x_1, x_2, \ldots, x_m \rangle$ *и* $Y = \langle y_1, y_2, \ldots, y_n \rangle$. *Тогда:*

1. *если* $x_m = y_n$, *то* $z_k = x_m = y_n$ *и* Z_{k-1} *является НОП для* X_{m-1} *и* Y_{n-1};
2. *если* $x_m \neq y_n$ *и* $z_k \neq x_m$, *то* Z *является НОП для* X_{m-1} *и* Y;
3. *если* $x_m \neq y_n$ *и* $z_k \neq y_n$, *то* Z *является НОП для* X_m *и* Y_{n-1}.

Доказательство. (1) Если $z_k \neq x_m$, то мы можем дописать $x_m = y_n$ в конец последовательности Z и получить общую подпоследовательность длины $k + 1$, что противоречит условию. Стало быть, $z_k = x_m = y_n$. Если у последовательностей X_{m-1} и Y_{n-1} есть более длинная (чем Z_{k-1}) общая подпоследовательность, то мы можем дописать к ней $x_m = y_n$ и получить общую подпоследовательность для X и Y, более длинную, чем Z — противоречие.

(2) Коль скоро $z_k \neq x_m$, последовательность Z является общей подпоследовательностью для X_{m-1} и Y. Так как Z — НОП для X и Y, то она тем более является НОП для X_{m-1} и Y.

(3) Аналогично (2). □

Мы видим, что НОП двух последовательностей содержит в себе наибольшую общую подпоследовательность их префиксов. Стало быть, задача о НОП обладает свойством оптимальности для подзадач. Сейчас мы убедимся, что перекрытие подзадач также имеет место.

Рекуррентная формула

Теорема 16.1 показывает, что нахождение НОП последовательностей $X = \langle x_1, x_2, \ldots, x_m \rangle$ и $Y = \langle y_1, y_2, \ldots, y_n \rangle$ сводится к решению либо одной, либо двух подзадач. Если $x_m = y_n$, то достаточно найти НОП последовательностей X_{m-1} и Y_{n-1} и дописать к ней в конце $x_m = y_n$. Если же $x_m \neq y_n$, то надо решить две подзадачи: найти

НОП для X_{m-1} и Y, а затем найти НОП для X и Y_{n-1}. Более длинная из них и будет служить НОП для X и Y.

Теперь сразу видно, что возникает перекрытие подзадач. Действительно, чтобы найти НОП X и Y, нам может понадобиться найти НОП X_{m-1} и Y, а также НОП X и Y_{n-1}; каждая из этих задач содержит подзадачу нахождения НОП для X_{m-1} и Y_{n-1}. Аналогичные перекрытия будут встречаться и далее.

Как и в задаче перемножения последовательности матриц, мы начнём с рекуррентного соотношения для стоимости оптимального решения. Пусть $c[i,j]$ обозначает длину НОП для последовательностей X_i и Y_j. Если i или j равно нулю, то одна из двух последовательностей пуста, так что $c[i,j] = 0$. Сказанное выше можно записать так:

$$c[i,j] = \begin{cases} 0 & \text{если } i = 0 \text{ или } j = 0, \\ c[i-1,j-1]+1 & \text{если } i,j > 0 \text{ и } x_i = y_j, \\ \max(c[i,j-1], c[i-1,j]), & \text{если } i,j > 0 \text{ и } x_i \neq y_j. \end{cases} \tag{16.5}$$

Вычисление длины НОП

Исходя из соотношения (16.5), легко написать рекурсивный алгоритм, работающий экспоненциальное время и вычисляющий длину НОП двух данных последовательностей. Но поскольку различных подзадач всего $\Theta(mn)$, лучше воспользоваться динамическим программированием.

Исходными данными для алгоритма LCS-Length служат последовательности $X = \langle x_1, x_2, \ldots, x_m \rangle$ и $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Числа $c[i,j]$ записываются в таблицу $c[0\,..\,m, 0\,..\,n]$ в таком порядке: сначала заполняется слева направо первая строка, затем вторая, и т. д. Кроме того, алгоритм запоминает в таблице $b[1\,..\,m, 1\,..\,n]$ *"происхождение"* $c[i,j]$: в клетку $b[i,j]$ заносится стрелка, указывающая на клетку с координатами $(i-1, j-1)$, $(i-1, j)$ или $(i, j-1)$, в зависимости от того, равно ли $c[i,j]$ числу $c[i-1,j-1]+1$, $c[i-1,j]$ или $c[i,j-1]$ (см. (16.5)). Результатами работы алгоритма являются таблицы c и b.

Рисунок 16.3 Таблицы c и b, созданные алгоритмом LCS-Length при $X = \langle A, B, C, B, D, A, B \rangle$ и $Y = \langle B, D, C, A, B, A \rangle$. В клетке с координатами (i, j) записаны число $c[i, j]$ и стрелка $b[i, j]$. Число 4 в правой нижней клетке есть длина НОП. При $i, j > 0$ значение $c[i, j]$ определяется тем, равны ли x_i и y_j, и вычисленными ранее значениями $c[i-1, j]$, $c[i, j-1]$ и $c[i-1, j-1]$. Путь по стрелкам, ведущий из $c[7, 6]$, заштрихован. Каждая косая стрелка на этом пути соответствует элементу НОП (эти элементы выделены).

```
LCS-Length($X, Y$)
$m \gets length[X]$
$n \gets length[Y]$
for $i \gets 1$ to $m$
    do $c[i, 0] \gets 0$
for $j \gets 0$ to $n$
    do $c[0, j] \gets 0$
for $i \gets 1$ to $m$
    do for $j \gets 1$ to $n$
           do if $x_i = y_j$
                 then $c[i, j] \gets c[i-1, j-1] + 1$
                      $b[i, j] \gets$ "$\nwarrow$"
                 else if $c[i-1, j] \geqslant c[i, j-1]$
                         then $c[i, j] \gets c[i-1, j]$
                              $b[i, j] \gets$ "$\uparrow$"
                         else $c[i, j] \gets c[i, j-1]$
                              $b[i, j] \gets$ "$\leftarrow$"
return $c$, $b$
```

На рис. 16.3 показана работа LCS-Length для $X = \langle A, B, C, B, D, A, B \rangle$ и $Y = \langle B, D, C, A, B, A \rangle$.

Алгоритм LCS-Length требует времени $O(mn)$: на каждую клетку требуется $O(1)$ шагов.

Построение НОП

Таблица b, созданная процедурой LCS-Length, позволяет быстро найти НОП последовательностей $X = \langle x_1, x_2, \ldots, x_m \rangle$ и $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Для этого надо пройти по пути, указанному стрелками, начиная с $b[m, n]$. Пройденная стрелка ↖ в клетке (i, j) означает, что $x_i = y_j$ входит в наибольшую общую подпоследовательность. Вот как это реализовано в рекурсивной процедуре Print-LCS (НОП для X и Y печатается при вызове Print-LCS$(b, X, \mathit{length}[X], \mathit{length}[Y])$):

```
Print-LCS($b, X, i, j$)
if $i = 0$ или $j = 0$
    then return
if $b[i, j] =$ "↖"
    then Print-LCS($b, X, i - 1, j - 1$)
         напечатать $x_i$
elseif $b[i, j] =$ "↑"
    then Print-LCS($b, X, i - 1, j$)
else Print-LCS($b, X, i, j - 1$)
```

Будучи применённой к таблице рис. 16.3, эта процедура напечатает $BCBA$. Время работы процедуры есть $O(m + n)$, поскольку на каждом шаге хотя бы одно из чисел m и n уменьшается.

Улучшение алгоритма

После того, как алгоритм разработан, нередко удаётся сделать его более экономным. В нашем примере можно обойтись без таблицы b. В самом деле, каждое из чисел $c[i, j]$ зависит от $c[i - 1, j]$, $c[i, j - 1]$ и $c[i - 1, j - 1]$. Зная $c[i, j]$, мы можем за время $O(1)$ выяснить, какая из этих трёх записей использовалась. Тем самым можно найти НОП за время $O(m+n)$ с помощью одной только таблицы c (в упражнении 16.3-2 мы попросим вас это сделать). При этом мы экономим $\Theta(mn)$ памяти. (Впрочем, асимптотика не меняется: объём таблицы c есть также $\Theta(mn)$.)

Если нас интересует только длина наибольшей общей подпоследовательности, то столько памяти не нужно: вычисление $c[i, j]$ затрагивает только две строки с номерами i и $i - 1$ (это не предел экономии: можно обойтись памятью на одну строку таблицы c плюс ещё чуть-чуть, см. упражнение 16.3-4). При этом, однако, саму подпоследовательность найти (за время $O(m + n)$) не удаётся.

Упражнения

16.3-1 Найдите НОП последовательностей $\langle 1,0,0,1,0,1,0,1\rangle$ и $\langle 0,1,0,1,1,0,1,1,0\rangle$.

16.3-2 Разработайте алгоритм, строящий НОП для $X = \langle x_1, x_2, \ldots, x_m\rangle$ и $Y = \langle y_1, y_2, \ldots, y_n\rangle$ за время $O(m+n)$, исходя только из таблицы c.

16.3-3 Разработайте рекурсивный вариант алгоритма LCS-Length с запоминанием ответов, требующий времени $O(mn)$.

16.3-4 Покажите, как можно вычислить длину НОП, используя память размера $2\min(m,n)+O(1)$ и храня лишь часть таблицы c. А как это сделать, используя память $\min(m,n)+O(1)$?

16.3-5 Разработайте алгоритм, находящий наибольшую возрастающую подпоследовательность данной последовательности из n чисел и работающий за время $O(n^2)$.

16.3-6* Разработайте алгоритм, решающий предыдущую задачу за время $O(n \log n)$. (Указание. Храним для каждого i наименьший из последних элементов возрастающих подпоследовательностей длины i. Как меняются эти числа при добавлении нового элемента?)

16.4 Оптимальная триангуляция многоугольника

Несмотря на свою геометрическую формулировку, эта задача очень близка к задаче о перемножении матриц.

Многоугольник (polygon) — это замкнутая кривая на плоскости, составленная из отрезков, называемых **сторонами** (sides) многоугольника. Точка, в которой сходятся две соседние стороны, называется **вершиной** (vertex). Несамопересекающийся многоугольник (только такие мы и будем, как правило, рассматривать) называется **простым** (simple). Множество точек плоскости, лежащих внутри простого многоугольника, называется **внутренностью** (interior) многоугольника, объединение его сторон называется его **границей** (boundary), а множество всех остальных точек плоскости называется его **внешностью** (exterior). Простой многоугольник называется **выпуклым** (convex), если для любых двух точек, лежащих внутри или на границе многоугольника, соединяющий их отрезок целиком лежит внутри или на границе многоугольника.

Выпуклый многоугольник можно задать, перечислив его вершины против часовой стрелки: многоугольник $P = \langle v_0, v_1, \ldots, v_{n-1}\rangle$

Рисунок 16.4 Две триангуляции выпуклого семиугольника. Каждая делит семиугольник на $7-2=5$ треугольников с помощью $7-3=4$ диагоналей.

имеет n сторон $\overline{v_0v_1}, \overline{v_1v_2}, \ldots, \overline{v_{n-1}v_n}$. Здесь v_n — то же самое, что v_0 (удобно нумеровать вершины n-угольника вычетами по модулю n).

Если v_i и v_j — две вершины, не являющиеся соседними, отрезок $\overline{v_iv_j}$ называется **диагональю** (chord) многоугольника. Диагональ $\overline{v_iv_j}$ разбивает многоугольник на два: $\langle v_i, v_{i+1}, \ldots, v_j\rangle$ и $\langle v_j, v_{j+1}, \ldots, v_i\rangle$. **Триангуляция** (triangulation) многоугольника — это набор диагоналей, разрезающих многоугольник на треугольники; сторонами этих треугольников являются стороны исходного многоугольника и диагонали триангуляции.

На рис. 16.4 изображены две триангуляции семиугольника. (Триангуляцию можно также определить как максимальное множество диагоналей, не пересекающих друг друга.)

Во всех триангуляциях n-угольника одно и то же число треугольников (сумма всех углов многоугольника равна произведению 180° и числа треугольников в триангуляции), а именно $n-2$. При этом используются $n-3$ диагонали (проводя диагональ, мы увеличиваем число частей на 1).

Задача об оптимальной триангуляции (optimal triangulation problem) состоит в следующем. Дан выпуклый многоугольник $P = \langle v_0, v_1, \ldots, v_{n-1}\rangle$ и весовая функция w, определённая на множестве треугольников с вершинами в вершинах P. Требуется найти триангуляцию, для которой сумма весов треугольников будет наименьшей.

Естественный пример весовой функции — функция

$$w(\triangle v_iv_jv_k) = |v_iv_j| + |v_jv_k| + |v_kv_i|,$$

где $|v_iv_j|$ обозначает (евклидово) расстояние между v_i и v_j. Мы построим алгоритм решения этой задачи, который применим для любой весовой функции.

Триангуляции и расстановки скобок

Существует удивительная связь между триангуляциями многоугольника и расстановками скобок (скажем, в произведении последовательности матриц). Проще всего объяснить эту связь с помощью деревьев.

Полной расстановке скобок соответствует так называемое **дерево разбора** (parse tree) выражения. На рис. 16.5а изображено дерево разбора для

$$((A_1(A_2A_3))(A_4(A_5A_6))). \tag{16.6}$$

В его листьях стоят матрицы-сомножители, а в вершинах — их произведения: в каждой вершине стоит произведение двух выражений, стоящих в её детях.

Триангуляцию выпуклого многоугольника $\langle v_0, v_1, \ldots, v_{n-1}\rangle$ также можно изобразить в виде дерева. Листьями его будут стороны многоугольника (кроме $\overline{v_0v_{n-1}}$). Остальные вершины — это диагонали триангуляции плюс сторона $\overline{v_0v_{n-1}}$; эту последнюю объявим корнем.

Построение дерева (на примере триангуляции рис. 16.5а) показано на рис. 16.5б. Для начала мы смотрим, в какой треугольник попал корень $\overline{v_0v_{n-1}}$. В нашем случае это $\triangle v_0v_3v_6$. Детьми корня будем считать две другие стороны этого треугольника. Триангуляция состоит из этого треугольника и двух триангуляций оставшихся частей ($\langle v_0, v_1, v_2, v_3\rangle$ и $\langle v_3, v_4, v_5, v_6\rangle$, рис. 16.5б), причём диагонали, являющиеся детьми корня, являются сторонами этих многоугольников ($\overline{v_0v_3}$ и $\overline{v_3v_6}$ на рис. 16.5б). Повторим для каждой из них ту же конструкцию: рассмотрим треугольник триангуляции нового многоугольника, содержащий выделенную сторону, две другие стороны этого треугольника объявим её детьми и т.д. В конце концов мы придём к бинарному дереву с $n-1$ листом. Действуя в обратном порядке, можно по бинарному дереву построить триангуляцию. Построенное соответствие между триангуляциями и бинарными деревьями является взаимно однозначным.

Вспоминая, что полные расстановки скобок в произведении n сомножителей находятся во взаимно однозначном соответствии с бинарными деревьями с n листьями, получаем взаимно однозначное соответствие между полными расстановками скобок в произведении n матриц и триангуляциями $(n+1)$-угольника. При этом матрица A_i в произведении $A_1A_2\ldots A_n$ соответствует стороне $\overline{v_{i-1}v_i}$, а диагональ $\overline{v_{i-1}v_j}$ $(1 \leqslant i < j \leqslant n)$ соответствует произведению $A_{i..j}$.

Это соответствие можно понять и без деревьев. Напишем на всех сторонах триангулированного многоугольника, кроме одной, по сомножителю. Далее поступаем так: если в треугольнике две стороны уже помечены, то на третьей мы пишем их произведение. На

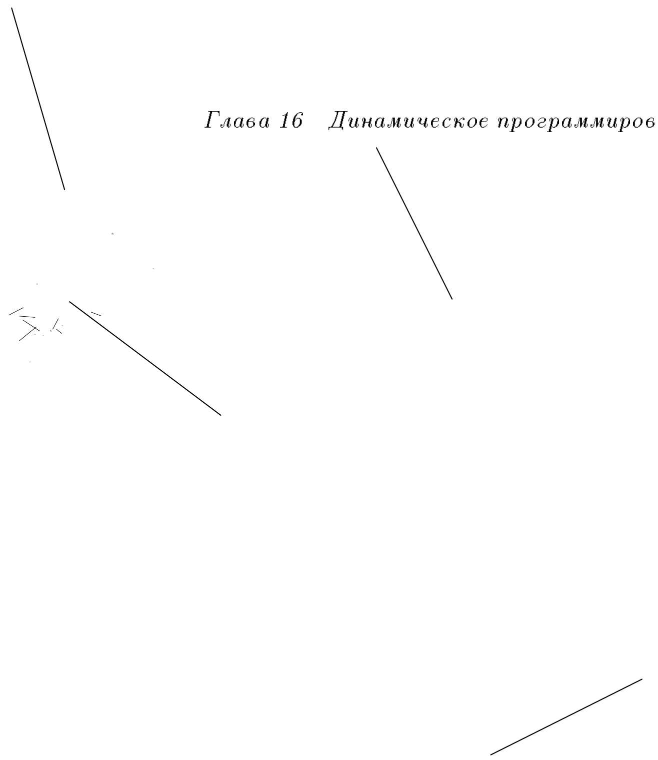

Рисунок 16.5 Деревья разбора. (а) Дерево разбора для $((A_1(A_2A_3))(A_4(A_5A_6)))$, соответствующее также триангуляции на рис. 16.4а. (б) Соответствие между триангуляцией многоугольника и бинарным деревом. Матрица A_i соответствует стороне $\overline{v_{i-1}v_i}$ $(i = 1, 2, \ldots, 6)$.
(в) Триангуляция и соответствующая ей расстановка скобок.

первоначально непомеченной стороне появится полная расстановка скобок (см. пример на рис. 16.5в).

Заметим, что задача о перемножении матриц является частным случаем задачи об оптимальной триангуляции. Пусть нам нужно вычислить $A_1A_2 \ldots A_n$, где A_i является $p_{i-1} \times p_i$-матрицей. Рассмотрим $(n+1)$-угольник $P = \langle v_0, v_1, \ldots, v_n \rangle$ и весовую функцию

$$w(\triangle v_iv_jv_k) = p_ip_jp_k.$$

Тогда стоимость триангуляции будет равна числу умножений при соответствующей расстановке скобок.

Хотя задача о перемножении матриц является лишь частным случаем задачи об оптимальной триангуляции, оказывается, что алгоритм Matrix-Chain-Order из раздела 16.1 легко приспособить для решения задачи о триангуляции. Надо только в его заголовке заменить p на v и строку 9 заменить на такую:

```
9            do $q \leftarrow m[i,k] + m[k+1,j] + w(\triangle v_{i-1}v_k v_j)$
```

В результате работы алгоритма $m[1,n]$ станет равным весу оптимальной триангуляции.

Строение оптимальной триангуляции

Докажем последнее утверждение. Пусть T — оптимальная триангуляция $(n+1)$-угольника $P = \langle v_0, v_1, \ldots, v_n\rangle$. Ребро $v_0 v_n$ входит в один из треугольников триангуляции. Пусть это треугольник $\triangle v_0 v_k v_n$, где $1 \leqslant k \leqslant n-1$. Тогда вес триангуляции T равен весу $\triangle v_0 v_k v_n$ плюс сумма весов триангуляций многоугольников $\langle v_0, v_1, \ldots, v_k\rangle$ и $\langle v_k, v_{k+1}, \ldots, v_n\rangle$. Следовательно, триангуляции указанных многоугольников обязаны быть оптимальными, и если оптимальные стоимости для таких многоугольников (при разных k) уже вычислены, то остаётся выбрать оптимальное значение k.

Рекуррентная формула

Другими словами, имеет место следующая рекуррентная формула. Пусть $m[i,j]$ — вес оптимальной триангуляции многоугольника $\langle v_{i-1}, v_i, \ldots, v_j\rangle$, где $1 \leqslant i < j \leqslant n$. Вес оптимальной триангуляции всего многоугольника равен $m[1,n]$. Будем считать, что *"двуугольники"* $\langle v_{i-1}, v_i\rangle$ имеют вес 0. Тогда $m[i,i] = 0$ для $i = 1, 2, \ldots, n$. Если $j - i \geqslant 1$, то у многоугольника $\langle v_{i-1}, v_i, \ldots, v_j\rangle$ имеется не менее трёх вершин, и нам необходимо найти минимум (по всем k из промежутка $i \leqslant k \leqslant j-1$) такой суммы: вес $\triangle v_{i-1} v_k v_j$, плюс вес оптимальной триангуляции $\langle v_{i-1}, v_i, \ldots, v_k\rangle$, плюс вес оптимальной триангуляции $\langle v_k, v_{k+1}, \ldots, v_j\rangle$. Поэтому

$$m[i,j] = \begin{cases} 0 & \text{при } i = j, \\ \min\limits_{i \leqslant k < j} \{m[i,k] + m[k+1,j] + w(\triangle v_{i-1} v_k v_j)\} & \text{при } i < j. \end{cases} \tag{16.7}$$

Единственное отличие этой формулы от формулы (16.2) — более общий вид весовой функции. Стало быть, алгоритм MATRIX-CHAIN-ORDER с указанными выше изменениями вычисляет вес оптимальной триангуляции; время работы $\Theta(n^3)$, объём используемой памяти $\Theta(n^2)$.

Упражнения

16.4-1 Докажите, что всякая триангуляция выпуклого n-угольника разбивает его на $n-2$ треугольника с помощью $n-3$ диагоналей.

16.4-2 Профессор предполагает, что для случая, когда вес треугольника равен его площади, алгоритм нахождения оптимальной триангуляции можно упростить. Не обманывает ли его интуиция?

16.4-3 Пусть весовая функция определена на множестве диагоналей многоугольника, и триангуляция считается оптимальной, если сумма весов входящих в неё диагоналей минимальна. Как свести эту задачу к разобранной нами (в которой веса приписывались не диагоналям, а треугольникам)?

16.4-4 Найдите оптимальную триангуляцию правильного восьмиугольника на евклидовой плоскости для случая, если весом треугольника считается его периметр.

Задачи

16-1 Битоническая евклидова задача коммивояжёра

Евклидова задача коммивояжёра (euclidean traveling-salesman problem) состоит в нахождении кратчайшего замкнутого пути, соединяющего данные n точек на плоскости (см. пример на рис. 16.6а, где $n = 7$). Эта задача является NP-полной, так что вряд ли её можно решить за полиномиальное время (см. главу 36).

Дж. Бентли предложил упростить задачу, рассматривая только **битонические пути** (bitonic tours), т.е. пути, начинающиеся в крайней левой точке, затем идущие слева направо до крайней правой точки, а затем возвращающиеся справа налево в исходную точку. (На рис. 16.6б изображён кратчайший битонический путь через те же семь точек). Эта задача проще: постройте алгоритм решения этой задачи, требующий времени $O(n^2)$. Вы можете считать, что абсциссы всех точек различны. (Указание: просматривая точки слева направо, храните для текущей точки X и для всех предыдущих точек Y длину кратчайшего пути, битонически соединяющего X с Y с проходом через крайнюю левую точку.)

16-2 Разбиение абзаца на строки

Абзац текста состоит из n слов длиной $l_1, l_2, \ldots, l_n$ (длина слова — число символов в нём). Считая, что все символы имеют равную ширину (как на пишущей машинке), мы хотим опти-

Рисунок 16.6 Семь точек на плоскости в узлах единичной решётки. (а) Кратчайший замкнутый путь (длины ≈ 24,88), обходящий эти точки. Этот путь не является битоническим. (б) Кратчайший битонический путь, обходящий те же точки, длина ≈ 25,58.

мальным образом разбить его на строки длиной не более M символов. Оптимальность при этом определяется так: посчитаем число ”*лишних*” пробелов в каждой строке (то есть посмотрим, на сколько длина строки меньше M, если между словами ставить по одному пробелу) и сложим кубы этих чисел для всех строк, кроме последней: чем больше эта сумма, тем хуже абзац. Используя динамическое программирование, разработайте алгоритм оптимального разбиения абзаца на строки; оцените время его работы и требуемый объём памяти.

16-3 Стоимость редактирования

Мы хотим преобразовать строку символов $x[1\,..\,m]$ в новую строку $y[1\,..\,n]$ с помощью следующих операций: перенос символа из исходной строки в новую, перенос двух соседних символов из исходной строки в новую с одновременной их перестановкой (”*транспозиция*”), добавление символа (справа) к новой строке, удаление первого символа из старой строки, замена (удаление символа из исходной строки и добавление другого символа в новую строку), наконец, удаление остатка старой строки.

Вот, например, как с помощью этих операций преобразовать строку `algorithm` в строку `altruistic`:

Операция	Новая строка	Старая строка
перенос `a`	`a`	`lgorithm`
перенос `l`	`al`	`gorithm`
замена `g` на `t`	`alt`	`orithm`
удаление `o`	`alt`	`rithm`
перенос `r`	`altr`	`ithm`
добавление `u`	`altru`	`ithm`
добавление `i`	`altrui`	`ithm`
добавление `s`	`altruis`	`ithm`
транспозиция `it` в `ti`	`altruisti`	`hm`
добавление `c`	`altruistic`	`hm`
удаление остатка	`altruistic`	

Каждой из операций (*"перенос"*, *"транспозиция"*, *"добавление"*, *"удаление"*, *"замена"* и *"удаление остатка"*) приписана стоимость (мы предполагаем, что стоимость замены меньше, чем суммарная стоимость добавления и удаления, иначе замена была бы лишней). В нашем примере стоимость преобразования равна

$$3 \cdot \text{стоимость переноса} + \text{стоимость замены} +$$
$$+ \text{стоимость удаления} + 4 \cdot \text{стоимость добавления} +$$
$$+ \text{стоимость транспозиции} + \text{стоимость удаления остатка}.$$

Пусть даны последовательности $x[1\,..\,m]$ и $y[1\,..\,n]$. Тогда **стоимость редактирования** (edit distance) есть наименьшая стоимость цепочки преобразований, переводящей x в y. Разработайте алгоритм, основанный на динамическом программировании, который вычисляет стоимость редактирования и находит оптимальную цепочку преобразований. Оцените время работы и объём используемой памяти.

16-4 Вечеринка в фирме

Профессору поручено устроить вечеринку в фирме *"ТОО КЛМН"*. Структура фирмы иерархична: отношение подчинённости задано деревом, корень которого — директор. Отдел кадров знает рейтинг каждого сотрудника (действительное число). Чтобы всем было весело, директор распорядился, чтобы никто не оказался на вечеринке вместе со своим непосредственным начальником.

а. Разработайте алгоритм, составляющий список приглашённых с наибольшим суммарным рейтингом.

б. Та же задача с дополнительным ограничением: директор должен быть в списке приглашённых.

16-5 Алгоритм Витерби

Динамическое программирование может быть применено к задаче распознавания речи. В качестве модели рассмотрим ориентированный граф $G = (V, E)$ и отображение σ, сопоставляющее с каждым ребром (u, v) звук $\sigma(u, v)$ из множества возможных звуков Σ. В графе выделена некоторая вершина v_0. Каждому пути, начинающемуся в v_0, соответствует последовательность звуков (элементов Σ).

а Разработайте эффективный алгоритм, восстанавливающий путь в графе по последовательности $s = \langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$ элементов Σ (если требуемого пути не существует, алгоритм должен выдать соответствующее сообщение). Оцените время работы алгоритма. (Указание: вам могут пригодиться некоторые идеи из главы 23).

Пусть теперь для каждого ребра $(u, v) \in E$ известна вероятность пройти по ребру из u в v (издать соответствующий звук); обозначим эту вероятность $p(u, v)$. Мы предполагаем, что для любой вершины u сумма вероятностей $p(u, v)$ по всем рёбрам, выходящим из этой вершины, равна единице. Вероятностью пути считаем произведение вероятностей, соответствующих его рёбрам (т.е. последовательные выборы считаем независимыми).

б. Модифицируйте построенный в (а) алгоритм так, чтобы он находил наиболее вероятный путь, соответствующий данной последовательности звуков. Оцените время работы модифицированного алгоритма.

Замечания

Систематическое изучение динамического программирования было начато Р. Беллманом в 1955 году [21], хотя некоторые приёмы такого рода были известны и ранее. Кстати, слово *"программирование"* (programming) в словосочетаниях *"динамическое программирование"* (dynamic programming), а также *"линейное программирование"* (linear programming) *не* означает составление программ для компьютера.

Ху и Шинг [106a, 106b] придумали работающий за время $O(n \log n)$ алгоритм для задачи о порядке перемножения матриц; они же указали соответствие между этой задачей и задачей об оптимальной триангуляции.

Алгоритм для нахождения наибольшей общей подпоследовательности за время $O(mn)$ относится, видимо, к фольклору. В работе [43] Кнут поставил вопрос, возможен ли для этой задачи субквадратичный алгоритм. Масек и Патерсон [143] показали, что возможен, найдя алгоритм, работающий за время $O(mn/\log n)$ при $n \leqslant m$ и фиксированном размере множества, из которого берутся члены последовательностей. Для случая, когда в последовательностях нет повторений, алгоритм с оценкой $O((n + m) \log(n + m))$ построен в статье Шимански [184]. Многие из этих результатов обобщаются на задачу о стоимости редактирования (задача 16-3).

17 Жадные алгоритмы

Для многих оптимизационных задач есть более простые и быстрые алгоритмы, чем динамическое программирование. В этой главе мы рассматриваем задачи, которые можно решать с помощью **жадных алгоритмов** (greedy algorithms). Такой алгоритм делает на каждом шаге локально оптимальный выбор, — в надежде, что итоговое решение также окажется оптимальным. Это не всегда так — но для многих задач такие алгоритмы действительно дают оптимум. Наш первый пример — простая, но не вполне тривиальная задача о выборе заявок (раздел 17.1). Далее (раздел 17.2) мы обсуждаем, для каких задач годятся жадные алгоритмы. В разделе 17.3 рассказывается о сжатии информации с помощью кодов Хаффмена, которые строятся жадным алгоритмом. Раздел 17.4 посвящён так называемым матроидам — комбинаторным объектам, связанным с жадными алгоритмами. Наконец, в разделе 17.5 мы применяем матроиды к задаче о расписании для заказов равной длительности со сроками и штрафами.

В следующих главах мы не раз встретимся с жадными алгоритмами (задача о минимальном покрывающем дереве, алгоритм Дейкстры поиска кратчайших путей из данной вершины, жадная эвристика Хватала для задачи о покрытии множества и другие). Минимальные покрывающие деревья — классический пример использования жадного алгоритма, так что параллельно с этой главой можно читать главу 24.

17.1 Задача о выборе заявок

Пусть даны n **заявок** на проведение занятий в одной и той же аудитории. Два разных занятия не могут перекрываться по времени. В каждой заявке указаны начало и конец занятия (s_i и f_i для i-й заявки). Разные заявки могут пересекаться, и тогда можно удовлетворить только одну из них. Мы отождествляем каждую заявку с промежутком $[s_i, f_i)$, так что конец одного занятия может совпадать с началом другого, и это не считается пересечением.

Формально говоря, заявки с номерами i и j **совместны** (compatible), если интервалы $[s_i, f_i)$ и $[s_j, f_j)$ не пересекаются (иными словами, если $f_i \leqslant s_j$ или $f_j \leqslant s_i$). **Задача о выборе заявок** (activity-selection problem) состоит в том, чтобы набрать максимальное количество совместных друг с другом заявок.

Жадный алгоритм работает следующим образом. Мы предполагаем, что заявки упорядочены в порядке возрастания времени окончания:

$$f_1 \leqslant f_2 \leqslant \ldots \leqslant f_n. \tag{17.1}$$

Если это не так, то можно отсортировать их за время $O(n \log n)$; заявки с одинаковым временем конца располагаем в произвольном порядке.

Тогда алгоритм выглядит так (f и s — массивы):

```
GREEDY-ACTIVITY-SELECTOR($s, f$)
1  $n \leftarrow length[s]$
2  $A \leftarrow \{1\}$
3  $j \leftarrow 1$
4  for $i \leftarrow 2$ to $n$
5       do if $s_i \geqslant f_j$
6             then $A \leftarrow A \cup \{i\}$
7                  $j \leftarrow i$
8  return $A$
```

Работа этого алгоритма показана на рис. 17.1. Множество A состоит из номеров выбранных заявок, j — номер последней из них; при этом

$$f_j = \max\{\, f_k : k \in A \,\}, \tag{17.2}$$

поскольку заявки отсортированы по временам окончания. Вначале A содержит заявку номер 1, и $j = 1$ (строки 2–3). Далее (цикл в строках 4–7) ищется заявка, начинающаяся не раньше окончания заявки номер j. Если таковая найдена, она включается в множество A и переменной j присваивается её номер (строки 6–7).

Алгоритм Greedy-Activity-Selector требует всего лишь $\Theta(n)$ шагов (не считая предварительной сортировки). Как и подобает жадному алгоритму, на каждом шаге он делает выбор так, чтобы остающееся свободным время было максимально.

Правильность алгоритма

Не для всех задач жадный алгоритм даёт оптимальное решение, но для нашей даёт. Убедимся в этом.

Теорема 17.1. *Алгоритм* Greedy-Activity-Selector *даёт набор из наибольшего возможного количества совместных заявок.*

i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14

Рисунок 17.1 Работа алгоритма Greedy-Activity-Selector для 11 заявок (таблица слева). Каждая строка на рисунке соответствует одному проходу цикла в строках 4–7. Серые заявки уже включены в A, белая сейчас рассматривается. Если левый конец белого прямоугольника левее правого конца правого серого (стрелка идёт влево), то заявка отвергается. В противном случае она добавляется к A.

Доказательство. Напомним, что заявки отсортированы по возрастанию времени окончания. Прежде всего докажем, что существует оптимальное решение задачи о выборе заявок, содержащее заявку номер 1 (с самым ранним временем окончания). В самом деле, если в каком-то оптимальном множестве заявок заявка номер 1 не содержится, то можно заменить в нём заявку с самым ранним временем окончания на заявку номер 1, что не повредит совместности заявок (ибо заявка номер 1 кончается ещё раньше, чем прежняя, и ни с чем пересечься не может) и не изменит их общего количества. Стало быть, можно искать оптимальное множество заявок A среди содержащих заявку номер 1: существует оптимальное решение, начинающееся с жадного выбора.

После того как мы договорились рассматривать только наборы, содержащие заявку номер 1, все несовместные с ней заявки можно выкинуть, и задача сводится к выбору оптимального набора заявок из множества оставшихся заявок (совместных с заявкой номер 1). Другими словами, мы свели задачу к аналогичной задаче с меньшим числом заявок. Рассуждая по индукции, получаем, что, делая на каждом шаге жадный выбор, мы придём к оптимальному решению. □

Упражнения

17.1-1 Задачу о выборе заявок можно решать с помощью динамического программирования, вычисляя последовательно (для $i = 1, 2, \ldots, n$) число m_i — максимально возможное число совместных заявок среди заявок с номерами $1, 2, \ldots, i$ (считая, что неравенство (17.1) выполнено). Сравните время работы такого алгоритма и алгоритма Greedy-Activity-Selector.

17.1-2 Пусть по-прежнему имеется множество заявок на проведение занятий (каждая заявка указывает начало и конец), но аудиторий сколько угодно. Нужно распределить заявки по аудиториям, используя как можно меньше аудиторий. Разработайте эффективный жадный алгоритм, решающий эту задачу.

Эта задача известна также как **задача о раскраске интервального графа** (interval-graph coloring problem). Рассмотрим граф, вершинами которого являются заявки, причём вершины соединены ребром тогда и только тогда, когда соответствующие заявки несовместны. Наименьшее количество цветов, необходимое для раскраски вершин графа таким образом, чтобы вершины одного цвета не были соединены, и есть наименьшее количество аудиторий, необходимое для выполнения всех заявок.

17.1-3 Для задачи о выборе заявок возможно несколько способов жадного выбора, и не все они годятся. Покажите на примере, что правила *"на каждом шаге выбирать заявку наименьшей длительности, совместную с уже выбранными"*, а также *"на каждом шаге выбирать заявку, совместную с наибольшим количеством остающихся"*, не годятся.

17.2 Когда применим жадный алгоритм?

Как узнать, даст ли жадный алгоритм оптимум применительно к данной задаче? Общих рецептов тут нет, но существуют две особенности, характерные для задач, решаемых жадными алгоритмами. Это принцип жадного выбора и свойство оптимальности для подзадач.

Принцип жадного выбора

Говорят, что к оптимизационной задаче применим **принцип жадного выбора** (greedy-choice property), если последовательность локально оптимальных (жадных) выборов дает глобально оптимальное решение. Различие между жадными алгоритмами и динамическим программированием можно пояснить так: на каждом шаге жадный алгоритм берёт *"самый жирный кусок"*, а потом уже пытается сделать наилучший выбор среди оставшихся, каковы бы они ни были; алгоритм динамического программирования принимает решение, просчитав заранее последствия для всех вариантов.

Как доказать, что жадный алгоритм даёт оптимальное решение? Это не всегда тривиально, но в типичном случае такое доказательство следует схеме, использованной в доказательстве теоремы 17.1. Сначала мы доказываем, что жадный выбор на первом шаге не закрывает пути к оптимальному решению: для всякого решения есть другое, согласованное с жадным выбором и не худшее первого. Затем показывается, что подзадача, возникающая после жадного выбора на первом шаге, аналогична исходной, и рассуждение завершается по индукции.

Оптимальность для подзадач

Говоря иными словами, решаемые с помощью жадных алгоритмов задачи обладают свойством **оптимальности для подзадач** (have optimal substructure): оптимальное решение всей задачи содержит в себе оптимальные решения подзадач. (С этим свойством мы уже встречались, говоря о динамическом программировании.) Например, при доказательстве теоремы 17.1 мы видели, что если A — оптимальный набор заявок, содержащий заявку номер 1, то $A' = A \setminus \{1\}$ — оптимальный набор заявок для меньшего множества заявок S', состоящего из тех заявок, для которых $s_i \geqslant f_1$.

Жадный алгоритм или динамическое программирование?

И жадные алгоритмы, и динамическое программирование основываются на свойстве оптимальности для подзадач, поэтому может возникнуть искушение применить динамическое программирование в ситуации, где хватило бы жадного алгоритма, или, напротив, применить жадный алгоритм к задаче, в которой он не даст оптимума. Мы проиллюстрируем возможные ловушки на примере двух вариантов классической оптимизационной задачи.

Дискретная задача о рюкзаке (0-1 knapsack problem) состоит в следующем. Пусть вор пробрался на склад, на котором хранится n вещей. Вещь номер i стоит v_i долларов и весит w_i килограммов (v_i и w_i — целые числа). Вор хочет украсть товара на максимальную сумму, причём максимальный вес, который он может унести в рюкзаке, равен W (число W тоже целое). Что он должен положить в рюкзак?

Непрерывная задача о рюкзаке (fractional knapsack problem) отличается от дискретной тем, что вор может дробить краденые товары на части и укладывать в рюкзак эти части, а не обязательно вещи целиком (если в дискретной задаче вор имеет дело с золотыми слитками, то в непрерывной — с золотым песком).

Обе задачи о рюкзаке обладают свойством оптимальности для подзадач. В самом деле, рассмотрим дискретную задачу. Вынув вещь номер j из оптимально загруженного рюкзака, получим решение задачи о рюкзаке с максимальным весом $W - w_j$ и набором из $n - 1$ вещи (все вещи, кроме j-й). Аналогичное рассуждение проходит и для непрерывной задачи: вынув из оптимально загруженного рюкзака, в котором лежит w килограммов товара номер j, весь этот товар, получим оптимальное решение непрерывной задачи, в которой максимальный вес равен $W - w$ (вместо W), а количество j-го товара равно $w_j - w$ (вместо w_j).

Хотя две задачи о рюкзаке и похожи, жадный алгоритм даёт оптимум в непрерывной задаче о рюкзаке и не даёт в дискретной. В самом деле, решение непрерывной задачи о рюкзаке с помощью жадного алгоритма выглядит так. Вычислим цены (в расчёте на килограмм) всех товаров (цена товара номер i равна v_i/w_i). Сначала вор берёт по максимуму самого дорогого товара; если весь этот товар кончился, а рюкзак не заполнен, вор берёт следующий по цене товар, затем следующий, и так далее, пока не наберёт вес W. Поскольку товары надо предварительно отсортировать по ценам, на что уйдёт время $O(n \log n)$, время работы описанного алгоритма будет $O(n \log n)$. В упражнении 17.2-1 мы попросим вас доказать, что описанный алгоритм действительно дает оптимум.

Рисунок 17.2 В дискретной задаче о рюкзаке жадная стратегия может не сработать. (а) Вор должен выбрать две вещи из трёх с тем, чтобы их суммарный вес не превысил 50 кг. (б) Оптимальный выбор — вторая и третья вещи; если положить в рюкзак первую, то выбор оптимальным не будет, хотя именно она дороже всех в расчёте на единицу веса. (в) Для непрерывной задачи о рюкзаке с теми же исходными данными выбор товаров в порядке убывания цены на единицу веса будет оптимален.

Чтобы убедиться в том, что аналогичный жадный алгоритм не обязан давать оптимум в дискретной задаче о рюкзаке, взгляните на рис. 17.2а. Грузоподъёмность рюкзака 50 кг, на складе имеются три вещи, весящие 10, 20 и 30 кг и стоящие 60, 100 и 120 долларов соответственно. Цена их в расчёте на единицу веса равна 6, 5 и 4. Жадный алгоритм для начала положит в рюкзак вещь номер 1; однако оптимальное решение включает предметы номер 2 и 3.

Для непрерывной задачи с теми же исходными данными жадный алгоритм, предписывающий начать с товара номер 1, даёт оптимальное решение (рис. 17.2в). В дискретной задаче такая стратегия не срабатывает: положив в рюкзак предмет номер 1, вор лишается возможности заполнить рюкзак *"под завязку"*, а пустое место в рюкзаке снижает цену наворованного в расчёте на единицу веса. При решении дискретной задачи о рюкзаке, чтобы решить, класть ли данную вещь в рюкзак, надо сравнить решение подзадачи, возникающей, если данная вещь заведомо лежит в рюкзаке, с подзадачей, возникающей, если этой вещи в рюкзаке заведомо нет. Тем самым дискретная задача о рюкзаке порождает множество перекрывающихся подзадач — типичный признак того, что может пригодиться динамическое программирование. И действительно, к дискретной задаче о рюкзаке оно применимо (см. упражнение 17.2-2).

Упражнения

17.2-1 Докажите, что для непрерывной задачи о рюкзаке выполнен принцип жадного выбора.

17.2-2 Разработайте основанный на динамическом программировании алгоритм для решения дискретной задачи о рюкзаке. Алгоритм должен работать за время $O(nW)$ (n — количество вещей, W — максимальный вес рюкзака).

17.2-3 Пусть в дискретной задаче о рюкзаке вещи можно упорядочить таким образом, чтобы одновременно выполнялись неравенства $w_1 \leqslant w_2 \leqslant \dots \leqslant w_n$ и $v_1 \geqslant v_2 \geqslant \dots \geqslant v_n$. Разработайте эффективный алгоритм для нахождения оптимального набора и докажите, что он правилен.

17.2-4 Профессор едет по шоссе из Петербурга в Москву, имея при себе карту с указанием всех стоящих на шоссе бензоколонок и расстояний между ними. Известно расстояние, которое может проехать машина с полностью заправленным баком. Разработайте эффективный алгоритм, позволяющий выяснить, на каких бензоколонках надо заправляться, чтобы количество заправок было минимально. (В начале пути бак полон.)

17.2-5 Дано n точек $x_1, x_2, \dots, x_n$ на координатной прямой; требуется покрыть все эти точки наименьшим числом отрезков длины 1. Разработайте эффективный алгоритм, решающий эту оптимизационную задачу.

17.2-6* Предполагая известным решение задачи 10-2, найдите решение непрерывной задачи о рюкзаке за время $O(n)$.

17.3 Коды Хаффмена

Коды Хаффмена широко используются при сжатия информации (в типичной ситуации выигрыш может составить от 20% до 90% в зависимости от типа файла). Алгоритм Хаффмена находит оптимальные коды символов (исходя из частоты использования этих символов в сжимаемом тексте) с помощью жадного выбора.

Пусть в файле длины 100 000 известны частоты символов (рис. 17.3). Мы хотим построить **двоичный код** (binary character code), в котором каждый символ представляется в виде конечной последовательности битов, называемой **кодовым словом** (codeword). При использовании **равномерного кода** (fixed-length code), в котором все кодовые слова имеют одинаковую длину, на каждый символ (из шести имеющихся) надо потратить как минимум три бита, например, $\mathtt{a} = 000, \mathtt{b} = 001, \dots, \mathtt{f} = 101$. На весь файл уйдет 300 000 битов — нельзя ли уменьшить это число?

Неравномерный код (variable-length code) будет экономнее, если часто встречающиеся символы закодировать короткими последо-

	a	b	c	d	e	f
Количество (в тысячах)	45	13	12	16	9	5
Равномерный код	000	001	010	011	100	101
Неравномерный код	0	101	100	111	1101	1100

Рисунок 17.3 Задача о выборе кода. В файле длиной 100 000 символов встречаются только латинские буквы от `a` до `f` (в таблице указано, по скольку раз каждая). Если каждую букву закодировать тремя битами, то всего будет 300 000 битов. Если использовать неравномерный код (нижняя строка), то достаточно 224 000 битов.

вательностями битов, а редко встречающиеся — длинными. Один такой код показан на рис. 17.3: буква `a` изображается однобитовой последовательностью 0, а буква `f` — четырёхбитовой последовательностью 1100. При такой кодировке на весь файл уйдёт

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224\,000$$

битов, что даёт около 25% экономии. (Далее мы увидим, что этот код будет оптимальным.)

Префиксные коды

Мы будем рассматривать только коды, в которых из двух последовательностей битов, представляющих различные символы, ни одна не является префиксом другой. Такие коды называются **префиксными кодами** (prefix codes; таков общепринятый термин, хотя логичнее было бы называть их "*беспрефиксными*" кодами). Можно показать (мы этого делать не будем), что для любого кода, обеспечивающего однозначное восстановление информации, существует не худший его префиксный код, так что мы ничего не теряем.

При кодировании каждый символ заменяется на свой код. Например, для неравномерного кода рис. 17.3 строка `abc` запишется как 0101100. Для префиксного кода декодирование однозначно и выполняется слева направо. Первый символ текста, закодированного префиксным кодом, определяется однозначно, так как кодирующая его последовательность не может быть началом кода какого-то иного символа. Найдя этот первый символ и отбросив кодировавшую его последовательность битов, мы повторяем процесс для оставшихся битов, и так далее. Например, строка 001011101 (при использовании неравномерного кода рис. 17.3) разбивается на части 0 0 101 1101 и декодируется как `aabe`.

Для эффективной реализации декодирования надо хранить информацию о коде в удобной форме. Одна из возможностей — представить код в виде двоичного дерева, листья которого соответствуют кодируемым символам. При этом путь от вершины дерева до кодируемого символа определяет кодирующую последовательность битов: поворот налево даёт 0, а поворот направо — 1.

Рисунок 17.4 Деревья, соответствующие двум кодам рис. 17.3. В каждом листе указан соответствующий символ и частота его использования в тексте. Во внутренних узлах указана сумма частот для листьев соответствующего поддерева. (а) Дерево, соответствующее равномерному коду $\mathtt{a} = 000, \ldots, \mathtt{f} = 100$. (б) Дерево, соответствующее оптимальному префиксному коду $\mathtt{a} = 0, \mathtt{b} = 101, \ldots, \mathtt{f} = 1100$.

На рис. 17.4 изображены деревья, соответствующие двум кодам рис. 17.3.

Оптимальному (для данного файла) коду всегда соответствует двоичное дерево, в котором всякая вершина, не являющаяся листом, имеет двоих детей (см. упражнение 17.3-1). В частности, равномерный код из нашего примера оптимальным быть не может, так как в соответствующем дереве (рис. 17.4а) есть вершина с одним ребёнком (коды некоторых символов начинаются с $10\ldots$, но ни один код символа не начинается с $11\ldots$). Такое свойство оптимального кода позволяет легко доказать, что дерево оптимального префиксного кода для файла, в котором используются все символы из некоторого множества C и только они, содержит ровно $|C|$ листьев, по одному на каждый символ, и ровно $|C| - 1$ узлов, не являющихся листьями.

Зная дерево T, соответствующее префиксному коду, легко найти количество битов, необходимое для кодирования файла. Именно, для каждого символа c из алфавита C пусть $f[c]$ обозначает число его вхождений в файл, а $d_T(c)$ — глубину соответствующего листа в дереве или, что то же самое, длину последовательности битов, кодирующей c. Тогда для кодирования файла потребуется

$$B(T) = \sum_{c \in C} f[c] d_T(c) \tag{17.3}$$

битов. Назовем это число **стоимостью** (cost) дерева T.

Построение кода Хаффмена

Хаффмен построил жадный алгоритм, который строит оптимальный префиксный код. Этот код называется **кодом Хаффмена** (Huffman code). Алгоритм строит дерево T, соответствующее оптимальному коду, снизу вверх, начиная с множества из $|C|$ листьев и делая $|C|-1$ *"слияний"*. Мы предполагаем, что для каждого символа $c \in C$ задана его частота $f[c]$. Для нахождения двух объектов, подлежащих слиянию, используется очередь с приоритетами Q, использующая частоты f в качестве рангов — сливаются два объекта с наименьшими частотами. В результате слияния получается новый объект (внутренняя вершина), частота которого считается равной сумме частот двух сливаемых объектов.

```
Huffman($C$)
1  $n \leftarrow |C|$
2  $Q \leftarrow C$
3  for $i \leftarrow 1$ to $n-1$
4       do $z \leftarrow$ Allocate-Node()
5          $x \leftarrow left[z] \leftarrow$ Extract-Min($Q$)
6          $y \leftarrow right[z] \leftarrow$ Extract-Min($Q$)
7          $f[z] \leftarrow f[x] + f[y]$
8          Insert($Q, z$)
9  return Extract-Min($Q$)
```

Работа этого алгоритма для нашего примера показана на рис. 17.5. Поскольку имеется всего 6 букв, первоначально очередь имеет размер $n = 6$, и для построения дерева нужно сделать 5 слияний. Префиксный код соответствует дереву, полученному в результате всех этих слияний.

В строке 2 алгоритма в очередь Q помещаются символы из алфавита C (с соответствующими частотами). В цикле (строки 3–8) повторяется $n-1$ раз следующая операция: из очереди изымаются две вершины x и y с наименьшими частотами $f[x]$ и $f[y]$, которые заменяются на одну вершину z с частотой $f[x]+f[y]$ и детьми x и y (кого из них считать левым ребёнком, а кого правым — неважно: код получится другой, но его стоимость будет та же). В конце в очереди остаётся один узел — корень построенного двоичного дерева. Ссылка на него возвращается в строке 9.

Оценим время работы алгоритма, считая, что очередь Q реализована в виде двоичной кучи (см. главу 7). Инициализацию Q в строке 2 можно провести за $O(n)$ операций с помощью процедуры Build-Heap из раздела 7.3. Цикл в строках 3–8 исполняется ровно $n-1$ раз; поскольку каждая операция с кучей требует времени $O(\log n)$, общее время будет $O(n \log n)$. Стало быть, время работы алгоритма Huffman для алфавита из n символов будет $O(n \log n)$.

Рисунок 17.5 Применение алгоритма Хаффмена к таблице частот рис. 17.3. Показаны последовательные состояния очереди (в порядке возрастания частот). На каждом шаге сливаются два поддерева с наименьшими частотами. Листья — прямоугольники, в которых символ и его частота разделены двоеточием. Внутренние вершины изображены кружками, в которых указаны суммы частот их детей. Ребро, идущее к левому ребёнку, помечено нулем, к правому — единицей. В результирующем дереве код любого символа написан на пути, ведущем из корня к этому символу. (а) Начальная стадия: $n = 6$ листьев, по одному на символ. (б)–(д) Промежуточные стадии. (е) Готовое дерево.

Правильность алгоритма Хаффмена

Чтобы доказать, что жадный алгоритм HUFFMAN действительно даёт оптимум, мы покажем, что для задачи об оптимальном префиксном коде выполнены принцип жадного выбора и свойство оптимальности для подзадач. Начнём с принципа жадного выбора.

Лемма 17.2. *Пусть в алфавите C каждый символ $c \in C$ имеет частоту $f[c]$. Пусть $x, y \in C$ — два символа с наименьшими частотами. Тогда для C существует оптимальный префиксный код, в котором последовательности битов, кодирующие x и y, имеют одинаковую длину и различаются только в последнем бите.*

Доказательство. Утверждение леммы будет выполнено, если листья, соответствующие x и y, будут братьями. Рассмотрим дерево T, соответствующее произвольному оптимальному префикс-

Рисунок 17.6 Доказательство леммы 17.2. Здесь b и c — наиболее удалённые от корня листья-братья, а x и y — листья с наименьшими частотами. Обмен x и b превращает дерево T в T', а обмен y и c превращает T' в T''. При каждой из перестановок стоимость дерева не возрастает.

ному коду, и покажем, что его можно модифицировать, не нарушая оптимальности, так, чтобы вышеуказанное условие выполнялось.

В самом деле, рассмотрим пару соседних (имеющих общего родителя) листьев в дереве T, находящуюся на максимальном расстоянии от корня. (Такие существуют: в оптимальном дереве все внутренние вершины имеют степень 2, и потому лист наибольшей глубины имеет брата.) Символы, стоящие в этих листьях (назовём их b и c) — не обязательно x и y, но заведомо имеют не меньшие частоты (поскольку x и y были двумя наиболее редкими символами). Не ограничивая общности, можно считать, что $f[x] \leqslant f[b]$ и $f[y] \leqslant f[c]$. Теперь поменяем местами в дереве T символы b и x (полученное дерево назовём T'), а затем символы c и y (полученное дерево назовём T''). После таких обменов x и y (см. пример на рис. 17.6) станут братьями, находящимися на максимальной глубине. Осталось убедиться, что при обменах стоимость дерева не возрастает и, следовательно, дерево T'' также является оптимальным. Другими словами, мы должны проверить, что $B(T) \geqslant B(T') \geqslant B(T'')$, где B — функция стоимости. В самом деле, стоимость определяется как сумма по всем листьям произведений частоты на глубину (17.3). При переходе от T к T' в этой сумме меняются только два слагаемых: $f[b]d_T(b)+f[x]d_T(x)$ заменяется на $f[b]d_{T'}(b)+f[x]d_{T'}(x)$, т.е. на $f[b]d_T(x)+f[x]d_T(b)$. Таким образом,

$$\begin{aligned}B(T)-B(T') &= f[b]d_T(b)+f[x]d_T(x)-f[b]d_T(x)-f[x]d_T(b) = \\ &= (f[b]-f[x])(d_T(b)-d_T(x)) \geqslant 0.\end{aligned}$$

Обе скобки неотрицательны: вспомним, что $f[x] \leqslant f[b]$ и что $d_T(b) \geqslant d_T(x)$, так как лист b был одним из наиболее удалённых от корня. Аналогичным образом $B(T') \geqslant B(T'')$, так что $B(T) \geqslant B(T'')$, и поэтому T'' также оптимально (а все наши неравенства являются равенствами). Лемма доказана. □

Доказанная лемма показывает, что построение оптимального дерева всегда можно начать со слияния двух символов с наименьшей частотой. Следующее наблюдение оправдывает название *"жадный"* для такого алгоритма. Назовём стоимостью слияния сумму частот сливаемых узлов. В упражнении 17.3-3 мы предложим вам доказать, что стоимость дерева равна сумме стоимостей всех слияний, необходимых для его построения. Стало быть, алгоритм HUFFMAN на каждом шаге выбирает слияние, наименее увеличивающее стоимость.

Теперь установим свойство оптимальности для подзадач.

Пусть фиксирован алфавит C и два символа x, y этого алфавита, а C' — алфавит, который получится из C, если выкинуть x и y и добавить новый символ z.

Рассмотрим кодовые деревья для C, в которых x и y (точнее, соответствующие им листья) являются братьями. Каждому такому дереву соответствует кодовое дерево для C', которое получится, если выбросить вершины x и y, а их общего родителя считать кодом символа z.

При этом соответствии каждому кодовому дереву для C' соответствует ровно два кодовых дерева для C (в одном из них x будет левым ребёнком, в другом — правым).

Пусть для каждого символа c из C фиксирована его частота $f[c]$. Определим частоты для символов из C', считая частотой символа z сумму $f[x]+f[y]$; для остальных символов частоты остаются теми же, что и в C. Тогда для кодовых деревьев (для обоих алфавитов) определены стоимости.

Лемма 17.3. *Стоимости соответствующих друг другу деревьев T и T' (при описанном соответствии) отличаются на величину $f[x]+f[y]$.*

Доказательство. Легко видеть, что $d_T(c) = d_{T'}(c)$ для всех $c \in C \setminus \{x, y\}$, а также что $d_T(x) = d_T(y) = d_{T'}(z) + 1$. Следовательно,

$$\begin{aligned} f[x]d_T(x) + f[y]d_T(y) &= (f[x]+f[y])(d_{T'}(z)+1) = \\ &= f[z]d_{T'}(z) + (f[x]+f[y]), \end{aligned}$$

откуда $B(T) = B(T') + f[x] + f[y]$. □

Эта лемма показывает, что выполнено свойство оптимальности для подзадач (оптимальное дерево T соответствует оптимальному дереву T' для меньшей задачи).

Из двух доказанных лемм легко следует

Теорема 17.4. *Алгоритм* HUFFMAN *строит оптимальный префиксный код.*

Доказательство. Лемма 17.2 показывает, что оптимальные кодовые деревья можно искать среди таких, у которых два наиболее редких символа (назовём из x и y) являются братьями. Им соответствуют деревья для алфавита C', в котором символы x и y слиты в один символ z. Считая частоту символа z равной сумме частот x и

y, можно применить лемму 17.3 и увидеть, что нам остаётся найти оптимальное кодовое дерево для алфавита C' и затем добавить к вершине z двух детей, помеченных символами x и y. Это и делает алгоритм Huffman. □

Упражнения

17.3-1 Докажите, что в бинарном дереве, соответствующем оптимальному префиксному коду, всякая вершина либо является листом, либо имеет двух детей.

17.3-2 Найдите код Хаффмена для алфавита, в котором частоты символов совпадают с первыми восемью числами Фибоначчи:

$$\mathtt{a}:1\quad \mathtt{b}:1\quad \mathtt{c}:2\quad \mathtt{d}:3\quad \mathtt{e}:5\quad \mathtt{f}:8\quad \mathtt{g}:13\quad \mathtt{h}:21.$$

Что будет, если в алфавите n символов, частоты которых совпадают с первыми n числами Фибоначчи?

17.3-3 Докажите, что стоимость двоичного дерева, соответствующего префиксному коду, можно вычислить следующим образом: для каждой вершины, не являющейся листом, найти сумму частот ее детей, и сложить все полученные числа.

17.3-4 Расположим символы алфавита в порядке убывания (невозрастания) частот. Докажите, что в оптимальном префиксном коде длины кодирующих эти символы последовательностей битов будут идти в неубывающем порядке.

17.3-5 Докажите, что оптимальный префиксный код для алфавита из n символов может быть представлен последовательностью из $2n-1+n\lceil\log_2 n\rceil$ битов. (Указание: для задания структуры дерева достаточно $2n-1$ битов.)

17.3-6 Обобщите алгоритм Хаффмена на тернарные коды (каждый символ кодируется последовательностью из цифр 0, 1 и 2). Коды, порождаемые вашим алгоритмом, должны быть оптимальны.

17.3-7 Пусть алфавит содержит $2^8 = 256$ символов, причём максимальная частота превосходит минимальную не более чем вдвое. Покажите, что для алфавита с такими частотами код Хаффмена не более эффективен, чем равномерный восьмибитовый код.

17.3-8 Профессор утверждает, что написанная им программа сжатия информации позволяет сжать любой файл длины 1000 (последовательность из тысячи 8-битовых байтов) хотя бы на один бит, после чего написанная им программа восстановления сможет восстановить исходный файл. Почему он неправ? (Указание: сравните количество возможных файлов с количеством сжатых файлов).

⋆ 17.4 Теоретические основы жадных алгоритмов

В этом разделе мы вкратце расскажем о красивом разделе комбинаторики, связанном с жадными алгоритмами, — теории матроидов. С помощью этой теории часто (хотя и не всегда: задачи из разделов 17.1 и 17.3 теорией матроидов не покрываются) удаётся установить, что данный жадный алгоритм даёт оптимум. Теория матроидов быстро развивается (см. ссылки в конце главы).

17.4.1 Матроиды

Матроидом (matroid) называется пара $M = (S, \mathcal{I})$, удовлетворяющая следующим условиям.

1. S — конечное непустое множество.
2. $\mathcal{I}$ — непустое семейство подмножеств S; входящие в $\mathcal{I}$ подмножества называют **независимыми** (independent). При этом должно выполняться такое свойство: из $B \in \mathcal{I}$ и $A \subseteq B$ следует $A \in \mathcal{I}$ (в частности, всегда $\emptyset \in \mathcal{I}$). Семейство $\mathcal{I}$, удовлетворяющее этому условию, называется **наследственным** (hereditary).
3. Если $A \in \mathcal{I}$, $B \in \mathcal{I}$ и $|A| < |B|$, то существует такой элемент $x \in B \setminus A$, что $A \cup \{x\} \in \mathcal{I}$. Это свойство семейства $\mathcal{I}$ называют **свойством замены** (exchange property).

Термин *"матроид"* принадлежит Хасслеру Уитни (Hassler Whitney). Он занимался **матричными матроидами** (matric matroids), у которых S — множество всех строк некоторой матрицы, и множество строк считается независимым, если эти строки линейно независимы в обычном смысле. (Легко показать, что действительно получается матроид, см. упр. 17.4-2.)

Другим примером является **графовый матроид** (graphic matroid) $(S_G, \mathcal{I}_G)$, строящийся по неориентированному графу G следующим образом: S_G совпадает со множеством рёбер графа, а $\mathcal{I}_G$ состоит из всех ацикличных (т.е. являющихся лесами) множеств рёбер. Графовые матроиды тесно связаны с задачей о минимальном покрывающем дереве, которую мы рассмотрим в главе 24.

Теорема 17.5. *Если $G = (V, E)$ — неориентированный граф, то $M_G = (S_G, \mathcal{I}_G)$ является матроидом.*

Доказательство. Так как подграф ацикличного графа ацикличен, множество $\mathcal{I}_G$ наследственно, и остаётся проверить свойство замены. В самом деле, пусть A и B — ацикличные подграфы G, причём $|B| > |A|$. Из теоремы 5.2 следует, что лес с k рёбрами является несвязным объединением $|V|-k$ деревьев, где $|V|$ — количество вершин (независимое доказательство: начнём с леса, состоящего из $|V|$ вершин и не имеющего ребёр, и будем по одному добавлять рёбра, не нарушая ацикличности; тогда добавление каждого ребра уменьшает количество связных компонент на единицу). Следовательно, лес A состоит из $|V| - |A|$ деревьев, а лес B — из $|V| - |B|$ деревьев. Поскольку $|V| - |B| < |V| - |A|$, лес B содержит такое дерево T, что две его вершины принадлежат разным связным компонентам леса A. Более того, поскольку T связно, оно должно содержать такое ребро (u, v), что u и v принадлежат разным связным компонентам леса A. Следовательно, добавление этого ребра к лесу A не может создать цикла, и его можно взять в качестве элемента x из определения матроида. ☐

(Графовые матроиды являются частным случаем матричных, если ребро графа рассматривать как формальную сумму его вершин с коэффициентами в поле вычетов по модулю 2, см. задачу 17-2б.)

Если $M = (S, \mathcal{I})$ — матроид, то элемент $x \notin A \in \mathcal{I}$ называется **независимым от** A (extension of A), если множество $A \cup \{x\}$ независимо. Например, в графовом матроиде ребро e независимо от леса A тогда и только тогда, когда его добавление к A не создаёт цикла.

Независимое подмножество в матроиде называется **максимальным** (maximal), если оно не содержится ни в каком большем независимом подмножестве. Часто бывает полезна следующая

Теорема 17.6. *Все максимальные независимые подмножества данного матроида состоят из одинакового числа элементов.*

Доказательство. Пусть A и B — максимальные независимые подмножества. Если, скажем, $|A| < |B|$, то из свойства замены вытекает существование такого $x \notin A$, что $A \cup \{x\}$ независимо — в противоречие с максимальностью. ☐

В качестве примера рассмотрим графовый матроид M_G, соответствующий связному графу G. Всякое максимальное независимое подмножество M_G должно быть деревом с $|V| - 1$ ребром, соединяющим все вершины G. Такое дерево называется **покрывающим (остовным) деревом** графа G (по-английски spanning tree).

Будем называть матроид $M = (S, \mathcal{I})$ **взвешенным** (weighted), если на множестве S задана весовая функция w со значениями во множестве положительных чисел. Функция w распространяется по аддитивности на все подмножества множества S; вес подмножества определяется как сумма весов его элементов: $w(A) = \sum_{x \in A} w(x)$. Пример: если M_G — графовый матроид, а $w(e)$ — длина ребра e, то $w(A)$ — сумма длин рёбер подграфа A.

17.4.2 Жадные алгоритмы для взвешенного матроида

Многие оптимизационные задачи, решаемые жадными алгоритмами, сводятся к задаче о нахождении в данном взвешенном матроиде $M = (S, \mathcal{I})$ независимого подмножества $A \subseteq M$ максимального веса. Независимое подмножество максимального веса называется **оптимальным** (optimal) подмножеством взвешенного матроида. Поскольку веса всех элементов положительны, оптимальное подмножество автоматически будет максимальным независимым подмножеством.

Например, **задача о наименьшем покрывающем дереве** (minimum-spanning-tree problem) состоит в следующем. Дан связный неориентированный граф $G = (V, E)$ и функция w из множества его рёбер во множество положительных чисел ($w(e)$ будем называть длиной ребра e). Требуется найти множество рёбер, соединяющих все вершины и имеющих наименьшую суммарную длину. Эту задачу можно рассматривать как частный случай задачи об оптимальном подмножестве взвешенного матроида. В самом деле, выберем число w_0, строго большее длин всех рёбер, и введем на графовом матроиде M_G веса по правилу $w'(e) = w_0 - w(e)$. Для всякого максимального независимого подмножества (т. е. покрывающего дерева) A имеем

$$w'(A) = (|V| - 1)w_0 - w(A),$$

где V — множество вершин графа. Стало быть, наименьшие покрывающие деревья для графа G — то же самое, что оптимальные подмножества в матроиде M_G с весовой функцией w'.

Задача о наименьшем покрывающем дереве подробно рассматривается в главе 24; сейчас мы приведём жадный алгоритм, находящий оптимальное подмножество A в любом взвешенном матроиде M. Если $M = (S, \mathcal{I})$, то мы пишем $S = S[M]$ и $\mathcal{I} = \mathcal{I}[M]$; весовая функция обозначается w.

```
GREEDY($M, w$)
1  $A \leftarrow \emptyset$
2  отсортировать $S[M]$ в порядке невозрастания весов
3  for $x \in S[M]$ (перебираем все $x$ в указанном порядке)
4      do if $A \cup \{x\} \in \mathcal{I}[M]$
5            then $A \leftarrow A \cup \{x\}$
6  return $A$
```

Алгоритм работает следующим образом. Полагаем $A = \emptyset$ (строка 1; пустое множество, как мы помним, всегда независимо) и перебираем элементы $S[M]$ в порядке убывания веса; если очередной элемент можно, не нарушая независимости, добавить к множеству A, то мы это делаем. Ясно, что полученное в результате множество будет независимым. Ниже мы покажем, что оно действительно будет иметь максимальный вес среди независимых подмножеств, а пока что оценим время работы алгоритма Greedy. Сортировка (строка 2) занимает время $O(n \log n)$, где $n = |S|$. Проверка независимости множества (строка 4) проводится n раз; если каждая такая проверка занимает время $f(n)$, то общее время работы будет $O(n \log n + nf(n))$.

Теперь покажем, что алгоритм Greedy действительно даёт оптимальное подмножество.

Лемма 17.7 (свойство жадного выбора для матроидов). *Пусть $M = (S, \mathcal{I})$ — взвешенный матроид с весовой функцией w. Пусть $x \in S$ — элемент наибольшего веса во множестве $\{\, y \in S : \{y\}$ независимо$\,\}$. Тогда x содержится в некотором оптимальном подмножестве $A \subseteq S$.*

Доказательство. Пусть B — какое-то оптимальное подмножество. Будем считать, что $x \notin B$, иначе доказывать нечего.

Положим $A' = \{x\}$. Это множество независимо по выбору x. Применяя $|B| - 1$ раз свойство замены, мы расширяем A' элементами из B и в конце концов построим независимое множество A, состоящее из x и $|B| - 1$ элемента множества B. Имеем $|A| = |B|$ (так что A максимально) и $w(A) = w(B) - w(y) + w(x)$, где y — единственный элемент B, не входящий в A. В то же время для всякого $y \in B$ множество $\{y\}$ независимо в силу свойства наследственности, так что $w(x) \geqslant w(y)$ по выбору x. Стало быть, $w(A) \geqslant w(B)$, и множество A также оптимально. Всё доказано. □

Далее, имеет место следующая очевидная

Лемма 17.8. *Если $M = (S, \mathcal{I})$ — матроид, $x \in S$ и $\{x\} \notin \mathcal{I}$, то $A \cup \{x\} \notin \mathcal{I}$ для всех $A \subseteq S$.*

Наша последняя лемма такова:

Лемма 17.9 (свойство оптимальности подзадач для матроидов). *Пусть $M = (S, \mathcal{I})$ — взвешенный матроид, и пусть $x \in S$ — некоторый его элемент, причём множество $\{x\}$ независимо. Тогда независимое множество наибольшего веса, содержащее x, является объединением $\{x\}$ и независимого множества наибольшего веса в матроиде $M' = (S', \mathcal{I}')$, где, по определению,*

$$S' = \{\, y \in S : \{x, y\} \in \mathcal{I} \,\},$$
$$\mathcal{I}' = \{\, B \subseteq S \setminus \{x\} : B \cup \{x\} \in \mathcal{I} \,\},$$

а весовая функция является ограничением на S' весовой функции для матроида M (в таких случаях говорят, что матроид M' получен из M **стягиванием** (contraction) элемента x).

Доказательство. Независимые в S множества, содержащие x, получаются добавлением элемента x к независимым в S' подмножествам. При этом их веса отличаются ровно на $w(x)$, так что оптимальные множества соответствуют оптимальным. $\square$

Теорема 17.10 (правильность жадного алгоритма для матроидов). *В результате работы алгоритма* Greedy, *применённого к взвешенному матроиду, получается оптимальное подмножество.*

Доказательство. Пусть $M = (S, \mathcal{I})$ — матроид с весовой функцией w. В силу леммы 17.8 мы можем не принимать во внимание элементы $x \in S$, для которых множество $\{x\}$ не независимо. Если $x \in S$ — первый из выбранных алгоритмом элементов, то лемма 17.7 показывает, что существует оптимальное подмножество $A \subseteq S$, содержащее x. Теперь, согласно лемме 17.9, достаточно найти оптимальное подмножество в матроиде M', полученном из M стягиванием элемента x (и добавить к нему x). Дальнейшая работа алгоритма в сущности и представляет собой обработку матроида M'. $\square$

[Более формальное рассуждение может быть таким. После любого числа итераций выполнен следующий инвариант цикла: (1) множество A независимо; (2) оптимальное множество можно искать среди множеств, являющихся объединением A с некоторыми из ещё не просмотренных элементов.]

Упражнения

17.4-1 Пусть S — конечное множество, k — натуральное число, $\mathcal{I}_k$ — семейство всех подмножеств S, содержащих не более k элементов. Покажите, что $(S, \mathcal{I}_k)$ — матроид.

17.4-2* Пусть T — вещественная матрица, S — множество её столбцов, и подмножество $A \subseteq S$ называется независимым, если оно линейно независимо в обычном смысле. Докажите, что получается матроид (называемый матричным, как мы уже говорили).

17.4-3* Пусть $(S, \mathcal{I})$ — матроид. Пусть $\mathcal{I}'$ — семейство всех таких подмножеств $A' \subseteq S$, для которых $S \setminus A'$ содержит некоторое максимальное независимое подмножество $A \subseteq S$. Покажите, что пара $(S, \mathcal{I}')$ является матроидом. Заметим, что максимальные независимые подмножества $(S, \mathcal{I}')$ суть дополнения максимальных независимых подмножеств $(S, \mathcal{I})$.

17.4-4* Пусть $S = S_1 \cup S_2 \cup \dots \cup S_k$ — разбиение конечного множества S на непересекающиеся непустые части. Положим $\mathcal{I} = \{\, A \subseteq S : |A \cap S_i| \leqslant 1$ для $i = 1, 2, \ldots, k \,\}$. Покажите, что пара $(S, \mathcal{I})$ является матроидом.

17.4-5 Пусть нам дан взвешенный матроид. Объясните, как надо модифицировать весовую функцию, чтобы свести задачу нахождения максимального независимого подмножества с наименьшим весом к задаче нахождения независимого подмножества с наибольшим весом.

⋆ 17.5 Задача о расписании

Интересным примером оптимизационной задачи, решаемой с помощью матроидов, является задача о расписании для заказов равной длительности с единственным исполнителем, сроками и штрафами. На первый взгляд эта задача кажется весьма запутанной, но жадный алгоритм дает неожиданно простое решение.

Итак, предположим, что имеется конечное множество S, состоящее из **заказов** (tasks), каждый из которых требует ровно одну единицу времени для своего выполнения. **Расписанием** (schedule) для S называется перестановка множества S, задающая порядок выполнения заказов: выполнение первого заказа начинается в момент времени 0 и заканчивается в момент 1, выполнение второго заказа начинается в момент времени 1 и заканчивается в момент 2, и т.д.

В **задаче о расписании для заказов равной длительности с единственным исполнителем, сроками и штрафами** (scheduling unit-time tasks with deadlines and penalties for a single processor) исходными данными являются:

- множество $S = \{1, 2, \ldots, n\}$, элементы которого мы называем заказами;
- последовательность из n целых чисел $d_1, d_2, \ldots, d_n$, называемых **сроками** (deadlines) ($1 \leqslant d_i \leqslant n$ для всех i, срок d_i относится к заказу номер i);
- последовательность из n неотрицательных чисел $w_1, w_2, \ldots, w_n$, называемых **штрафами** (penalties) (если заказ номер i не выполнен ко времени d_i, взимается штраф w_i).

Требуется найти расписание для S, при котором сумма штрафов будет наименьшей.

Заказ номер i **просрочен** (late) для данного расписания, если его выполнение завершается позже момента d_i, в противном случае заказ считается **выполненным в срок** (early). Всякое расписание можно, не меняя суммы штрафов, модифицировать таким образом, чтобы все просроченные заказы стояли в нём после выполненных в срок. В самом деле, если просроченный заказ y идёт раньше выполненного в срок заказа x, то можно поменять их местами в расписании, и статус обоих заказов при этом не изменится.

Более того, каждое расписание можно, не меняя суммы штрафов, представить в **каноническом виде** (canonical form), а именно, так, чтобы все просроченные заказы стояли после выполненных в срок, а сроки для выполненных в срок заказов шли в неубывающем порядке. В самом деле, как мы видели, можно считать, что все просроченные заказы стоят после выполненных в срок. Если теперь выполнение непросроченных заказов номер i и j завершается в моменты времени k и $k+1$ соответственно, но при этом $d_j < d_i$, то можно поменять заказы местами, и оба по-прежнему не будут просрочены (коль скоро $d_j \geqslant k+1$, то и подавно $d_i > d_j \geqslant k+1$; заказу же номер j вообще ничего не грозит, поскольку по новому расписанию его будут выполнять раньше, чем по старому). Конечное число таких обменов приведёт расписание к каноническому виду.

Таким образом, задача о расписании сводится к нахождению множества A, состоящего из заказов, которые не будут просрочены: как только это множество найдено, для составления расписания достаточно расположить заказы из множества A в порядке возрастания сроков, а после них в произвольном порядке поставить остальные заказы.

Будем говорить, что подмножество $A \subseteq S$ является **независимым** (independent), если для этого множества заказов можно составить расписание, по которому все заказы будут выполнены в срок. Обозначим через $\mathcal{I}$ семейство всех независимых подмножеств множества S.

Как выяснить, будет ли данное подмножество $A \subseteq S$ независимым? Для каждого $t = 1, 2, \ldots, n$ обозначим через $N_t(A)$ количество заказов из множества A, для которых срок не превосходит t.

Лемма 17.11. *Для всякого подмножества $A \subseteq S$ следующие три условия эквивалентны:*

1. *множество A независимо,*
2. *для всех $t = 1, 2, \ldots, n$ имеем $N_t(A) \leqslant t$,*
3. *если расположить заказы из множества A в порядке неубывания сроков, то все заказы будут выполнены в срок.*

Доказательство. Если $N_t(A) > t$ для некоторого t, то заказов, которые должны быть выполнены за первые t единиц времени, больше t и один из них непременно будет просрочен. Таким образом, (1) влечёт (2). Если выполнено (2), то i-й по порядку срок окончания заказа не меньше i, и при расстановке заказов в этом порядке все сроки будут соблюдены. Следовательно, (2) влечёт (3). Импликация (3) $\Rightarrow$ (1) очевидна. □

Условие (2) является удобным критерием независимости множества заказов (см. упражнение 17.5-2).

Минимизировать сумму штрафов за просроченные заказы — всё равно, что максимизировать сумму невыплаченных штрафов, то

	Заказ						
	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Рисунок 17.7 Пример задачи о расписании для заказов равной длительности с единственным исполнителем, сроками и штрафами.

есть штрафов, ассоциированных с выполненными в срок заказами. Следующая теорема показывает, что эту оптимизационную задачу можно решить с помощью жадного алгоритма:

Теорема 17.12. *Пусть S — множество заказов равной длительности со сроками, а $\mathcal{I}$ — семейство независимых множеств заказов. Тогда пара $(S, \mathcal{I})$ является матроидом.*

Доказательство. Очевидно, каждое подмножество независимого множества также независимо, и остаётся проверить выполнение свойства замены. Пусть A и B — независимые множества, причём $|B| > |A|$. Будем сравнивать числа $N_t(B)$ и $N_t(A)$ при различных t. При $t = n$ первое число больше; уменьшая t, дождёмся момента, когда они сравняются, и назовём его k (если этого не произойдет до самого конца, считаем $k = 0$). При этом $N_k(A) = N_k(B)$ (если $k > 0$) и $N_{k+1}(B) > N_{k+1}(A)$. Следовательно, есть хотя бы один заказ $x \in B \setminus A$ со сроком $k + 1$. Положим $A' = A \cup \{x\}$. Если $t \leqslant k$, то $N_t(A') = N_t(A) \leqslant t$ в силу независимости множества A; если $t > k$, то $N_t(A') = N_t(A) + 1 \leqslant N_t(B) \leqslant t$ в силу независимости множества B; стало быть, множество A' независимо по лемме 17.11, и для пары $(S, \mathcal{I})$ выполнено свойство замены. Всё доказано. ☐

Из доказанной теоремы следует, что для нахождения оптимального множества A независимых заказов можно воспользоваться жадным алгоритмом, а затем составить расписание, начинающееся с заказов из множества A, расставленных в порядке возрастания сроков — это и будет решением задачи о расписании. Если пользоваться алгоритмом Greedy, то время работы будет $O(n^2)$, так как в процессе работы этого алгоритма надо сделать n проверок независимости множества, и каждая такая проверка требует $O(n)$ операций (упр. 17.5-2). Более быстрый алгоритм приведён в задаче 17-3.

На рис. 17.7 приведён пример задачи о расписании. Жадный алгоритм отбирает заказы 1, 2, 3 и 4, затем отвергает заказы 5 и 6 и отбирает заказ 7. Оптимальное расписание:

$$(2, 4, 1, 3, 7, 5, 6).$$

Сумма штрафов равна $w_5 + w_6 = 50$.

Упражнения

17.5-1 Решите задачу о расписании для семи заказов, в которой сроки те же, что на рис. 17.7, но каждый штраф w_i заменён на $80 - w_i$.

17.5-2 Как, используя условие (2) из леммы 17.11, выяснить за время $O(|A|)$, будет ли данное множество заказов A независимым?

Задачи

17-1 Сдача с доллара

Пусть требуется набрать сумму в n центов, используя наименьшее количество монет.

а Опишите жадный алгоритм, набирающий n центов с помощью монет достоинством в 25, 10, 5 и 1 цент. [Именно такие монеты используются в США.] Докажите, что алгоритм находит оптимальное решение.

б Пусть в нашем распоряжении имеются монеты достоинством $c^0, c^1, \ldots, c^k$ центов, где $c > 1$ и $k \geqslant 1$ — целые числа. Докажите, что жадный алгоритм даст в этом случае оптимальное решение.

в Приведите пример набора типов монет, для которого жадный алгоритм оптимума не даст.

17-2 Ацикличные подграфы

а Пусть $G = (V, E)$ — неориентированный граф. Покажите, что пара $(E, \mathcal{I})$, где $A \in \mathcal{I}$ тогда и только тогда, когда множество A ациклично, является матроидом.

б **Матрицей инцидентности** (incidence matrix) неориентированного графа $G = (V, E)$ называется $|V| \times |E|$-матрица M, в которой M_{ve} равно единице, если вершина v инцидентна ребру e, и нулю в противном случае. (Столбец, соответствующий ребру, содержит ровно две единицы, соответствующие концам этого ребра.) Докажите, что набор столбцов этой матрицы линейно независим [над полем вычетов из двух элементов] тогда и только тогда, когда соответствующий набор ребер ацикличен. Пользуясь результатом упражнения 17.4-2, докажите теперь другим способом, что пара $(E, \mathcal{I})$ является матроидом.

в Пусть в неориентированном графе $G = (V, E)$ для каждого ребра $e \in E$ задан неотрицательный вес $w(e)$. Разработайте эффективный алгоритм для нахождения ацикличного подмножества множества E с наибольшей суммой весов рёбер.

г Пусть $G = (V, E)$ — ориентированный граф, и пусть $\mathcal{I}$ — семейство всех ацикличных подмножеств множества рёбер (в данном случае слово "*ацикличный*" означает "*не содержащий ориентированных циклов*"). Приведите пример, когда пара $(E, \mathcal{I})$ не будет матроидом. Какое из условий в определении матроида будет нарушено?

д **Матрица инцидентности** для ориентированного графа $G = (V, E)$ — это $|V| \times |E|$-матрица M, в которой M_{ve} равно -1, если вершина v является началом ребра e, равно 1, если вершина v является концом ребра e, и равно нулю в остальных случаях. Докажите, что множество рёбер графа, соответствующее линейно независимому (над $\mathbb{R}$) набору столбцов матрицы инцидентности, не содержит ориентированного цикла.

е В упражнении 17.4-2 мы доказали, что линейно независимые наборы столбцов данной матрицы образуют матроид. В свете этого результата, нет ли противоречия между утверждениями пунктов (г) и (д) упражнения?

17-3 Ещё о расписаниях

Рассмотрим следующий алгоритм для решения задачи из раздела 17.5 (оптимальное расписание для заказов равной длительности с единственным исполнителем, сроками и штрафами). Будем перебирать заказы в порядке убывания штрафов и заполнять расписание так: если для заказа номер j существует хотя бы одно свободное место в расписании, позволяющее выполнить его не позднее требуемого срока d_j, то поставим его на самое позднее из таких мест; в противном случае поставим его на самое позднее из свободных мест.

а Докажите, что этот алгоритм даёт оптимум.

б Воспользуйтесь представлением непересекающихся множеств с помощью леса, описанным в разделе 22.3, для эффективной реализации вышеописанного алгоритма (можете считать, что заказы уже расположены в порядке убывания штрафов). Оцените время работы алгоритма.

Замечания

Дополнительные сведения о жадных алгоритмах и матроидах можно найти у Лоулера [132] или Пападимитриу и Стайглица [154].

Первой работой по комбинаторной оптимизации, содержащей жадный алгоритм, была работа Эдмондса [62], датированная 1971 годом, хотя само понятие матроида было введено в 1935 году в статье Уитни [200].

Задачами этой главы занимались многие авторы — Гаврил [80] (задача о выборе заявок), Лоулер [132], Горовиц и Сахни [105], Брассар и Братли [33] (задача о расписании).

Коды Хаффмена были изобретены в 1952 году [107]. Обзор методов сжатия информации (по состоянию на 1987 год) дали Лелевер и Хиршберг [136].

Корте и Ловас [127, 128, 129, 130] создали теорию гридоидов (greedoids), являющуюся обобщением теории, изложенной в разделе 17.4.

18 Амортизационный анализ

Амортизационный анализ применяется для оценки времени выполнения нескольких операций с какой-либо структурой данных (например, стеком). Чтобы оценить время выполнения какой-либо последовательности операций, достаточно умножить максимальную длительность операции на общее число операций. Иногда, однако, удается получить более точную оценку времени работы (или, что равносильно, среднего времени выполнения одной операции), используя тот факт, что во многих случаях после длительных операций несколько следующих операций выполняются быстро. Оценки такого рода называются **амортизационным анализом** (amortized analysis) алгоритма. Подчеркнем, что оценка, даваемая амортизационным анализом, не является вероятностной: это оценка среднего времени выполнения одной операции для худшего случая.

[При амортизационном анализе каждой операции присваивается некоторая **учётная стоимость** (amortized cost), которая может быть больше или меньше реальной длительности операции. При этом должно выполняться следующее условие: для любой последовательности операций фактическая суммарная длительность всех операций (предполагается, что до выполнения операций структура данных находится в начальном состоянии — например, стек пуст) не превосходит суммы их учётных стоимостей. Если это условие выполнено, то говорят, что учётные стоимости присвоены корректно. Заметим, что для одной и той же структуры данных и одних и тех же алгоритмов выполнения операций можно корректно назначить учётные стоимости несколькими различными способами.]

В первых трёх разделах этой главы рассказывается о трёх основных методах амортизационного анализа. В разделе 18.1 речь идет о методе группировки: мы можем оценить стоимость n операций в худшем случае и установить, что она не превосходит $T(n)$). После этого можно объявить, что учётная стоимость любой операции, независимо от ее длительности, равна $T(n)/n$.

В разделе 18.2 рассказывается о методе предоплаты. При этом методе амортизационного анализа различным операциям могут присваиваться различные учётные стоимости. У некоторых операций учётная стоимость объявляется выше реальной. Когда выполняется такая операция, остаётся резерв, который считается хранящимся в определенном месте структуры данных. Этот резерв используется для доплаты за операции, учётная стоимость которых ниже фактической.

В методе потенциалов, обсуждаемом в разделе 18.3, резерв не связывается с какими-то конкретными объектами, а является функцией текущего состояния структуры данных. Эта функция называется *"потенциалом"*, и учётные стоимости должны быть с ней согласованы.

Мы иллюстрируем эти три метода на двух примерах. Первый из них — стек, снабженный дополнительной операцией Multipop, удаляющей из стека несколько элементов одновременно. Второй пример — двоичный счетчик, снабженный единственной операцией Increment (увеличить на единицу).

Заметим, что понятие потенциала используется для анализа алгоритма; в самой программе нет необходимости вычислять и хранить его значение.

Идеи, связанные с амортизационным анализом, полезно иметь в виду при разработке алгоритмов. Пример такого рода дан в разделе 18.4 (таблица переменного размера).

18.1 Метод группировки

Метод группировки (aggregate method) состоит в следующем: для каждого n оценивается время $T(n)$, требуемое на выполнение n операций (в худшем случае). Время в расчёте на одну операцию, то есть отношение $T(n)/n$, объявляется учётной стоимостью одной операции. При этом учётные стоимости всех операций оказываются одинаковыми (при двух других методах амортизационного анализа это будет не так).

Операции со стеком

Мы применим метод группировки для анализа стека с одной дополнительной операцией. В разделе 11.1 мы ввели две основные операции со стеком:

Push(S, x) добавляет элемент x к стеку S;

Pop(S) удаляет вершину стека S (и возвращает её).

Каждая из этих операций выполняется за время $O(1)$. Следовательно, выполнение n операций Push и Pop требует времени $\Theta(n)$.

Ситуация становится более интересной, если добавить операцию Multipop(S, k), которая удаляет k элементов из стека (если в стеке

Рисунок 18.1 Действие операции Multipop на стеке S. (а) Начальное состояние стека. Multipop$(S, 4)$ удаляет из стека верхние четыре элемента, и стек приходит в состояние (б). Если теперь применить операцию Multipop$(S, 7)$, то стек станет пуст (в), так как перед применением этой операции в стеке было менее семи элементов.

содержится менее k элементов, то удаляется всё, что там есть). Её можно реализовать так (Stack-Empty возвращает true тогда и только тогда, когда стек пуст):

```
Multipop($S, k$)
while not Stack-Empty($S$) and $k \neq 0$
    do Pop($S$)
       $k \leftarrow k - 1$
```

Пример работы операции Multipop приведен на рис. 18.1.

При применении операции Multipop(S, k) к стеку, содержащему s элементов, будет произведено $\min(s, k)$ операций Pop. Поэтому (имеется в виду фактическая стоимость, а не учётная) время ее работы пропорционально $\min(s, k)$.

Считая, что операции Push, Pop имеют единичную стоимость (имеется в виду фактическая стоимость, а не учётная), а операция Multipop имеет стоимость $\min(s, k)$, оценим суммарную стоимость n операций, применённых к изначально пустому стеку. Стоимость любой из операций не превосходит n (самая дорогая операция Multipop стоит не более n, поскольку за n операций в стеке не наберётся более n элементов). Следовательно, суммарная стоимость n операций есть $O(n^2)$. Эта оценка, однако, слишком груба. Чтобы получить точную оценку, заметим, что, коль скоро первоначально стек был пуст, общее число реально выполняемых операций Pop (включая те из них, что вызываются из процедуры Multipop) не превосходит общего числа операций Push, а это последнее не превосходит n. Стало быть, стоимость любой последовательности из n операций Push, Pop и Multipop, примененных к пустому стеку, есть $O(n)$. Тем самым можно объявить, что учётная стоимость каждой из операций есть $O(n)/n = O(1)$.

Подчеркнём ещё раз, что наша оценка не является вероятностной: стоимость (в расчёте на одну операцию) оценивается для худшего случая.

counter value — значение счетчика,
total cost — стоимость

Рисунок 18.2 Состояния восьмибитового двоичного счетчика в процессе выполнения 16 последовательных операций Increment. Заштрихованы биты, значения которых изменятся при следующей операции Increment. В правой графе показана общая стоимость всех операций по установке или очистке битов, необходимых, чтобы досчитать до данного числа. Заметьте, что в строке номер k эта стоимость не превосходит $2k$.

Двоичный счётчик

В этом разделе мы применим метод группировки к анализу k-битного двоичного счётчика. Счётчик реализован как массив битов $A[0\,..\,k-1]$ и хранит двоичную запись числа $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$ (так что $A[0]$ — младший бит). Первоначально $x = 0$, т.е. $A[i] = 0$ для всех i. Определим операцию Increment (увеличить на 1 по модулю 2^k) так:

```
Increment($A$)
1  $i \leftarrow 0$
2  while $i < length[A]$ and $A[i] = 1$
3      do $A[i] \leftarrow 0$
4         $i \leftarrow i + 1$
5  if $i < length[A]$
6     then $A[i] \leftarrow 1$
```

По существу тот же алгоритм используется в реальных компьютерах (каскадное сложение — см. раздел 29.2.1). На рис. 18.2 изображены состояния двоичного счетчика при 16 последовательных применениях операции Increment, начиная от 0 до 16. Увеличение счетчика на единицу происходит следующим образом: все начальные единичные биты в массиве A становятся нулями (цикл в строках 2–4), а следующий непосредственно за ними нулевой бит (если таковой есть) устанавливается в единицу. Стоимость операции Increment линейно зависит от общего количества битов, подвергшихся очистке или установке. Отсюда сразу получается грубая оценка: поскольку в худшем случае (массив A состоит из одних единиц) меняются k битов, требуется $O(nk)$ операций, чтобы сосчитать от 0 до n.

Чтобы получить более точную оценку, заметим, что не каждый раз значения всех k битов меняются. В самом деле, младший бит $A[0]$ меняется при каждом исполнении операции Increment (см. рис. 18.2). Следующий по старшинству бит $A[1]$ меняется только через раз: при отсчете от нуля до n этот бит меняется $\lfloor n/2 \rfloor$ раз. Далее, $A[N]$ меняется только каждый четвертый раз, и так далее: если $0 \leqslant i \leqslant \lg n$, то в процессе счета от 0 до n бит $A[i]$ меняется $\lfloor n/2^i \rfloor$ раз, а если $i > \lfloor \lg n \rfloor$, то бит i вообще не меняется. Стало быть, общее количество операций очистки и установки битов равно

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n.$$

Тем самым увеличение двоичного счётчика от 0 до n требует в худшем случае $O(n)$ операций, причём константа не зависит от k (и равна 2); учётную стоимость каждой операции можно считать равной $O(n)/n = O(1)$ (константа опять же не зависит от k).

Упражнения

18.1-1 Предположим, что мы разрешили не только операцию Multipop, но и Multipush, добавляющую к стеку произвольное количество элементов. Можно ли теперь считать учётную стоимость каждой операции равной $O(1)$?

18.1-2 Определим для k-битового двоичного счетчика операцию Decrement (вычитание единицы по модулю 2^k). Покажите, последовательность из n операций Increment и Decrement в худшем случае требует времени $\Theta(nk)$.

18.1-3 Рассмотрим последовательность операций, в которой стоимость операции номер i равна i, если i является степенью двойки, и 1 в противном случае. Оцените среднюю стоимость одной операции в последовательности из n операций.

18.2 Метод предоплаты

При проведении амортизационного анализа по **методу предоплаты** (accounting method) каждой операции присваивается своя учётная стоимость, причем эти стоимости могут быть как больше, так и меньше реальных. Если учетная стоимость превосходит реальную, разность рассматривается как **резерв** (credit), который считается связанным с каким-то объектом структуры данных и рассматривается как предоплата за будущую обработку этого объекта. За счет этого резерва компенсируется разница между учетной и реальной стоимостью для тех операций, у которых учетная стоимость ниже реальной.

Мы должны выбирать учётные стоимости так, чтобы фактическая стоимость не превосходила суммы учетных стоимостей, то есть чтобы резерв оставался неотрицательным в любой момент работы.

Операции со стеком

Проиллюстрируем метод предоплаты на уже известном примере со стеком. Напомним, что реальные стоимости операций со стеком мы считаем такими:

Push	1,
Pop	1,
Multipop	$\min(s, k)$,

где k — количество элементов, удаляемых из стека, s — размер стека. Присвоим этим операциям такие учётные стоимости:

Push	2,
Pop	0,
Multipop	0.

В нашем примере учётные стоимости всех операций постоянны; возможны случаи, когда эти стоимости непостоянны и даже имеют различные асимптотики для разных операций.

Покажем теперь, что выбранные нами учетные стоимости позволяют полностью покрыть реальную стоимость операций со стеком. Будем считать, что единицей стоимости операций является доллар. Воспользуемся аналогией между стеком и стопкой тарелок (см. разд. 11.1). Первоначально ни одной тарелки в стопке нет (мы начинаем с пустого стека). Когда мы добавляем тарелку к стопке, мы должны заплатить 1 доллар за операцию Push (это ее реальная цена). Один доллар остается у нас в резерве, ведь учётная цена операции Push равна двум долларам. Будем всякий раз класть этот резервный доллар на соответствующую тарелку. Тогда в каждый момент на каждой тарелке в нашей стопке будет лежать по долларовой бумажке.

Доллар, лежащий на тарелке, — это предоплата за удаление тарелки из стопки. Когда мы производим операцию Pop, мы за нее ничего дополнительно не платим (учётная стоимость Pop равна нулю), а расплачиваемся за удаление тарелки лежащим на ней резервным долларом. Точно так же мы имеем возможность объявить операцию Multipop бесплатной: если надо удалить k тарелок, за удаление каждой из них мы расплатимся лежащим на ней долларом. Стало быть, избыточная плата, которую мы берем за операцию Push, покрывает расходы на операции Pop и Multipop. (стоимость любой последовательности из n операций Push, Pop и Multipop не превосходит суммарной учётной стоимости).

Двоичный счётчик

Решим таким же способом задачу о двоичном счётчике. Нам надо провести амортизационный анализ последовательности из n операций Increment (в исходном состоянии в счетчике записан нуль). Мы уже отмечали, что время работы этой операции пропорционально суммарному количеству установок и очисток битов. Будем считать, что каждая установка или очистка стоит 1 доллар.

Установим такие учётные стоимости: 2 доллара за установку бита, 0 за очистку. При каждой установке бита одним из двух долларов учетной цены будем расплачиваться за реальные затраты на эту установку, а доллар, остающийся в резерве, будем прикреплять к установленному биту. Поскольку первоначально все биты были нулевыми, в каждый момент к каждому ненулевому биту будет прикреплен резервный доллар. Стало быть, за очистку любого бита ничего платить нам не придется: мы расплатимся за нее долларовой бумажкой, прикреплённой к этому биту в момент его установки.

Теперь легко определить учётную стоимость операции Increment: поскольку каждая такая операция требует не более одной установки бита, ее учётную стоимость можно считать равной 2 долларам. Следовательно, фактическая стоимость n последовательных операций Increment, начинающихся с нуля, есть $O(n)$, поскольку она не превосходит суммы учётных стоимостей.

Упражнения

18.2-1 Предположим, что мы работаем со стеком, максимальный размер которого равен k, причем после каждых k операций делается резервная копия стека. Покажите, что суммарная стоимость n операций со стеком (включая резервное копирование) есть $O(n)$, выбрав подходящие учётные стоимости.

18.2-2 Сделайте упражнение 18.1-3 с помощью метода предоплаты.

18.2-3 Добавим к системе операций с двоичным счётчиком операцию Reset (очистить все биты). Реализуйте счетчик на базе вектора битов таким образом, чтобы стоимость последовательности n операций Increment и Reset, примененных к счётчику, в котором первоначально находился нуль, была равна $O(n)$, причём константа не должна зависесть от k — числа битов в счётчике. (Указание: храните номер старшего ненулевого бита).

18.3 Метод потенциалов

Метод потенциалов (potential method) является обобщением метода предоплаты: здесь резерв не распределяется между отдельными элементами структуры данных (например, отдельными битами двоичного счетчика), а является функцией состояния структуры данных в целом. Эта функция называется *"потенциальной энергией"*, или *"потенциалом"*.

Общая схема метода потенциалов такова. Пусть над структурой данных предстоит произвести n операций, и пусть D_i — состояние структуры данных после i-й операции (в частности, D_0 — исходное состояние). **Потенциал** (potential) представляет собой функцию Φ из множества возможных состояний структуры данных во множество действительных чисел; эта функция называется ещё **потенциальной функцией** (potential function). Пусть c_i — реальная стоимость i-й операции. **Учетной стоимостью** (amortized cost) i-й операции объявим число

$$\widehat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}), \tag{18.1}$$

т.е. сумму реальной стоимости операции плюс приращение потенциала в результате выполнения этой операции. Тогда суммарная учетная стоимость всех операций равна

$$\sum_{i=1}^{n} \widehat{c}_i = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0). \tag{18.2}$$

Если нам удалось придумать функцию Φ, для которой $\Phi(D_n) \geqslant \Phi(D_0)$, то суммарная учетная стоимость даст верхнюю оценку для реальной стоимости последовательности n операций. Не ограничивая общности, можно считать, что $\Phi(D_0) = 0$ (см. упр. 18.3-1).

Говоря неформально, если разность потенциалов $\Phi(D_i) - \Phi(D_{i-1})$ положительна, то учетная стоимость i-й операции включает в себя резерв (*"предоплату"* за будущие операции); если же эта разность отрицательна, то учетная стоимость i-й операции меньше реальной, и разница покрывается за счет накопленного к этому моменту потенциала.

Учётные стоимости (и тем самым оценки на реальную стоимость), рассчитанные с помощью метода потенциалов, зависят от выбора потенциальной функции, а сам этот выбор является делом творческим.

Операции со стеком

Проиллюстрируем метод потенциалов на примере знакомой нам задачи о стеке с операциями Push, Pop и Multipop. В качестве потенциальной функции Φ возьмем количество элементов в стеке. Поскольку мы начинаем с пустого стека, имеем

$$\Phi(D_i) \geqslant 0 = \Phi(D_0).$$

Стало быть, сумма учётных стоимостей оценивает сверху реальную стоимость последовательности операций.

Найдем теперь учётные стоимости индивидуальных операций. Так как операция Push имеет реальную стоимость 1 и к тому же увеличивает потенциал на 1, её учётная стоимость равна $1+1=2$; операция Multipop, удаляющая из стека m элементов, имеет стоимость m, но уменьшает потенциал на m, так что её учётная стоимость равна $m-m=0$; точно так же равна нулю и учетная стоимость операции Pop. Коль скоро учётная стоимость каждой операции не превосходит 2, стоимость последовательности n операций, начинающихся с пустого стека, есть $O(n)$.

Двоичный счетчик

Проанализируем с помощью метода потенциалов двоичный счётчик. В качестве потенциальной функции возьмём количество единиц в счетчике.

Найдем учётную стоимость операции Increment. Пусть b_i — количество единиц в счетчике после i-й операции, и пусть i-я операция Increment очищает t_i битов; тогда $b_i \leqslant b_{i-1} - t_i + 1$ (кроме очистки битов, Increment может ещё установить в единицу не более одного бита). Стало быть, реальная стоимость i-го Increment'а не превосходит $t_i + 1$, а его учетная стоимость есть

$$\widehat{c}_i \leqslant (t_i + 1) + (b_i - b_{i-1}) \leqslant 2.$$

Если отсчёт начинается с нуля, то $\Phi(D_0) = 0$, так что $\Phi(D_i) \geqslant \Phi(D_0)$ для всех i, сумма учётных стоимостей оценивает сверху сумму реальных стоимостей, и суммарная стоимость n операций Increment есть $O(n)$ с константой (двойкой), не зависящей от k (размера счётчика).

Метод потенциалов позволяет разобраться и со случаем, когда отсчет начинается не с нуля. В этом случае из (18.2) вытекает, что

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \widehat{c}_i - \Phi(D_n) + \Phi(D_0), \qquad (18.3)$$

откуда, принимая во внимание, что $\widehat{c}_i \leqslant 2$ для всех i, получаем, что

$$\sum_{i=1}^{n} c_i \leqslant 2n - b_n + b_0.$$

Поскольку $b_0 \leqslant k$, для достаточно длинных последовательностей операций ($n = \Omega(k)$ операций Increment) реальная стоимость оценивается как $O(n)$, причём константа в O-записи не зависит ни от k, ни от начального значения счетчика.

Упражнения

18.3-1 Пусть потенциальная функция Φ такова, что $\Phi(D_i) \geqslant \Phi(D_0)$ для всех i, но $\Phi(D_0) \neq 0$. Покажите, что существует потенциальная функция Φ', для которой $\Phi'(D_0) = 0$, $\Phi'(D_i) \geqslant 0$ для всех i, и учётные стоимости, рассчитанные с помощью функции Φ', совпадают с учётными стоимостями, рассчитанными с помощью Φ.

18.3-2 Сделайте упражнение 18-1.3 с помощью метода потенциалов.

18.3-3 Пусть наша структура данных — двоичная куча с операциями Insert и Extract-Min, работающими за время $O(\lg n)$ в худшем случае, где n — количество элементов. Придумайте потенциальную функцию Φ, для которой учётная стоимость операции Insert есть $O(\lg n)$, учётная стоимость операции Extract-Min есть $O(1)$, и сумма учётных стоимостей последовательности операций оценивает сверху реальную стоимость последовательности операций.

18.3-4 Пусть последовательность из n операций Push, Pop и Multipop применяется к стеку, содержащему s_0 элементов, и приводит к стеку, содержащему s_n элементов. Какова суммарная фактическая стоимость этих операций?

18.3-5 Пусть в начальном состоянии двоичный счетчик содержит b единиц, и пусть $n = \Omega(b)$. Докажите, что стоимость n операций Increment есть $O(n)$ с константой, не зависящей от b.

18.3-6 Как реализовать очередь на базе двух стеков (упражнение 11.1-6) таким образом, чтобы учетная стоимость операций Enqueue и Dequeue была $O(1)$?

18.4 Динамические таблицы

При работе с таблицей переменного размера бывает желательно запрашивать память блоками: когда в таблице нет места для нового элемента, мы запрашиваем место для таблицы большего размера и копируем записи из старой таблицы в новую. При этом новый размер таблицы выбирается с запасом, чтобы не пришлось немедленно расширять её ещё раз. Напротив, если при удалении записей таблица становится почти пустой, то имеет смысл скопировать её остаток в таблицу меньшего размера и освободить память, занятую старой таблицей. В этом разделе мы покажем, как можно динамически расширять и сжимать таблицы, если мы хотим, чтобы учётная стоимость операций добавления и удаления записей к таблице была $O(1)$, (хотя реальная стоимость добавлений или удалений, требующих расширения или сжатия таблицы, может быть велика). При этом мы будем следить, чтобы коэффициент заполнения таблицы (отношение числа использованных ячеек к общему размеру) был не слишком мал.

Будем считать, что динамическая таблица поддерживает операции Table-Insert (добавить к таблице) и Table-Delete (удалить из таблицы). При добавлении записи к таблице мы занимаем в таблице одну **ячейку** (slot), при удалении записи одна ячейка освобождается. Как конкретно реализованы сама таблица и эти операции, нас в данный момент не интересует: это может быть стек (раздел 11.1), куча (раздел 7.1), хеш-таблица (глава 12), или, наконец, массив или набор массивов (раздел 11.3).

Как мы уже говорили (см. также главу 12 о хешировании), Именно, **коэффициентом заполнения** (load factor) таблицы T назовем число $\alpha(T)$, равное отношению числа заполненных ячеек к размеру таблицы (общему числу ячеек), если знаменатель отличен от нуля. Для вырожденной таблицы (не содержащей ни одной ячейки) коэффициент заполнения мы считаем равным единице. Если определенный таким образом коэффициент заполнения ограничен снизу положительным числом, то доля свободных ячеек в таблице ограничена сверху числом, меньшим единицы.

18.4.1 Расширение таблицы

Для начала рассмотрим динамическую таблицу, в которую можно добавлять записи, а нельзя удалять. Память для таблицы выделяется в виде массива ячеек. Таблица заполнена, когда в ней нет свободных ячеек (иными словами, когда коэффициент заполнения равен 1)[1]. Будем считать, что наша операционная система, как и большинство современных, снабжена подсистемой управления памятью, позволяющей при необходимости выделять новые блоки памяти и освобождать занятые. Имея это в виду, при попытке добавить запись к заполненной таблице мы будем предварительно **расширять** (expand) таблицу следующим образом: выделим память под таблицу увеличенного размера, скопируем имеющиеся записи из старой таблицы в новую, после чего добавим запись уже к новой таблице.

Как показывает опыт, разумно увеличивать таблицу вдвое в момент переполнения. Если мы только добавляем записи к таблице, то при такой стратегии коэффициент заполнения не может стать меньше 1/2 (если он не был таким в самом начале), т.е. пустые ячейки не будут занимать более половины таблицы.

Вот как выглядит алгоритм Table-Insert, основанный на этих принципах ($table[T]$ –указатель на блок памяти, в котором размещена таблица T, $num[T]$ — количество записей в таблице, $size[T]$ — размер таблицы, т.е. количество ячеек; первоначально $size[T] = num[T] = 0$):

```
Table-Insert$(T, x)$
 1  if $size[T] = 0$
 2     then выделить блок $table[T]$ с одной ячейкой
 3          $size[T] \leftarrow 1$
 4  if $num[T] = size[T]$
 5     then выделить блок памяти $new\text{-}table$ с $2 \cdot size[T]$ ячейками
 6          скопировать все записи из $table[T]$ в $new\text{-}table$
 7          освободить память, занятую $table[T]$
 8          $table[T] \leftarrow new\text{-}table$
 9          $size[T] \leftarrow 2 \cdot size[T]$
10  записать $x$ в $table[T]$
11  $num[T] \leftarrow num[T] + 1$
```

Проанализируем время работы этого алгоритма. Собственно запись в таблицу происходит в нём только в строках 6 и 10. Стоимость операции *"записать элемент в таблицу"* примем за единицу. Будем считать, что время работы всей процедуры Table-Insert пропорционально количеству операций *"записать в таблицу"*; тем самым мы подразумеваем, что затраты на выделение памяти под таблицу с одной ячейкой (строка 2) — величина ограниченная, и что затраты на выделение и освобождение памяти в строках 5 и 7 — величина не большего порядка, чем затраты на

---

[1] В некоторых приложениях (например, в хеш-таблицах с открытой адресацией) разумно считать таблицу заполненной уже в тот момент, когда коэффициент заполнения превышает некоторую константу, меньшую единицы (см. упр. 18-4.2).

переписывание из одной таблицы в другую (строка 6).

Пусть теперь мы применили n операций Table-Insert к таблице, не содержащей первоначально ни одной ячейки. Какова стоимость i-й операции, после которой в таблице будет i элементов? Обозначим эту стоимость через c_i. Если в таблице есть место (или если $i = 1$), то $c_i = 1$ (мы записываем новый элемент в таблицу один раз в строке 10. Если же места в таблице нет и добавление записи к таблице сопровождается расширением таблицы, то $c_i = i$: одна единица стоимости идет на запись нового элемента в расширенную таблицу (строка 10), и ещё $i - 1$ единица пойдет на копирование старой таблицы в новую (строка 6). Стало быть, $c_i \leqslant n$ при всех i, и грубая оценка стоимости n операций Table-Insert есть $O(n^2)$. Можно, однако, эту оценку улучшить, если проследить, сколь часто происходит расширение таблицы. В самом деле, расширение происходит только в том случае, когда $i-1$ является степенью двойки, так что

$$\sum_{i=1}^{n} c_i \leqslant n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j < n + 2n = 3n.$$

Стало быть, стоимость n операций Table-Insert не превосходит $3n$, и учётную стоимость каждой такой операции можно считать равной 3.

Понять, откуда берется коэффициент 3, можно с помощью метода предоплаты. Именно, при выполнении каждой операции Table-Insert(T, x) будем вносить по 3 доллара, при этом один доллар немедленно израсходуем на оплату операции *"записать x в table$[T]$"* в строке 10, ещё один доллар прикрепим к элементу x, а оставшийся отдадим одному из элементов, уже записанных в таблицу (из тех, что ещё не получили своего доллара) Этими долларами будет оплачиваться перенос записей из старой таблицы в расширенную. При этом всякий раз, когда таблицу необходимо расширять, каждый из её элементов будет иметь по доллару. В самом деле, пусть при предыдущем расширении таблица имела размер m и была расширена до $2m$. С тех пор мы добавили m элементов, при этом и добавленные элементы, и каждый из m старых элементов получили по доллару. Таким образом, следующее расширение таблицы уже оплачено.

Можно проанализировать алгоритм Table-Insert и с помощью метода потенциалов. Чтобы это сделать, надо придумать такую потенциальную функцию, которая равна нулю сразу после расширения таблицы и нарастает по мере расширения таблицы (становясь равной размеру таблицы в тот момент, когда свободных ячеек не остаётся). Этим условиям удовлетворяет, например, функция

$$\Phi(T) = 2 \cdot num[T] - size[T]. \tag{18.4}$$

Начальное значение потенциала равно нулю, и поскольку в любой момент не менее половины таблицы заполнено, значение $\Phi(T)$ все-

Рисунок 18.3 Размер таблицы ($size_i$), число записей (num_i) и потенциал ($\Phi_i = 2\cdot num_i - size_i$) как функция от числа операций Table-Insert, обозначенного буквой i. График *num* — сплошная тонкая линия, график *size* — сплошная жирная линия, график Φ — пунктир. Непосредственно перед расширением таблицы Φ возрастает до num_i, чтобы оплатить это расширение. Сразу после расширения потенциал падает до нуля, но тут же и возрастает до 2 (за счет добавления записи после расширения).

гда неотрицательно. Стало быть, сумма учётных стоимостей операций Table-Insert (определённых по методу потенциалов как фактическая стоимость плюс изменение потенциала) оценивает сверху суммарную фактическую стоимость.

Найдем учётные стоимости. Для этого обозначим через $size_i$, num_i и Φ_i размер, количество записей и потенциал таблицы после i-й операции Table-Insert. Если эта операция не сопровождалась расширением таблицы, то $size_i = size_{i-1}$, $num_i = num_{i-1}+1$ и $c_i = 1$, откуда

$$\widehat{c}_i = c_i + \Phi_i - \Phi_{i-1} = 3.$$

Если же i-я операции Table-Insert сопровождалась расширением таблицы, то $size_i/2 = size_{i-1} = num_{i-1}$ и $c_i = num_i$, откуда опять-таки

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi_i - \Phi_{i-1} = \\ &= num_i + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) = \\ &= num_i + (2num_i - (2num_i - 2)) - (2(num_i - 1) - (num_i - 1)) = \\ &= num_i + 2 - (num_i - 1) = 3.\end{aligned}$$

На рис. 18.3 изображена зависимость величин num_i, $size_i$ и Φ_i от i. Обратите внимание, как накапливается потенциал к моменту расширения таблицы.

18.4.2 Расширение и сокращение таблицы

Теперь займемся операцией Table-Delete (удалить из таблицы). Помимо удаления ненужной записи из таблицы, желательно **сократить** (contract) таблицу, как только коэффициент заполнения станет слишком низок. Общая стратегия тут та же, что и при расширении таблицы: как только число записей падает ниже некоторого предела, мы выделяем память под новую таблицу меньшего размера, копируем в меньшую таблицы все записи из большей, после чего возвращаем операционной системе память, занимавшуюся старой таблицей. Мы построим алгоритм Table-Delete со следующими свойствами:

- коэффициент заполнения ограничен снизу положительной константой,
- учётная стоимость операций Table-Insert и Table-Delete есть $O(1)$.

При этом мы по-прежнему будем считать, что стоимость алгоритма определяется количеством операций *"записать в таблицу"* (строки 6 и 10 алгоритма Table-Insert) и противоположных ей операций *"убрать из таблицы"*; стоимость каждой из этих элементарных операций примем за единицу.

Первое, что приходит в голову, — удваивать размер таблицы всякий раз, когда надо добавить запись к заполненной таблице, и сокращать размер таблицы вдвое всякий раз, когда в результате удаления записи коэффициент заполнения падает ниже $1/2$. При такой стратегии коэффициент заполнения никогда не опустится ниже $1/2$, но, к сожалению, стоимости операций оказываются слишком велики. В самом деле, пусть n — натуральное число, являющееся степенью двойки. Предположим, что сначала мы добавили к пустой (не содержащей ячеек) таблице $n/2+1$ записей, а затем провели ещё $n/2-1$ операцию в такой последовательности: два удаления, затем два добавления, затем опять два удаления, опять два добавления, и т. д. После $n/2+1$ добавлений размер таблицы станет равным n (и заполнено будет чуть больше половины ячеек); после двух удалений его придется сократить до $n/2$; после следующих двух добавлений размер опять возрастет до n, после двух удалений — сократится до $n/2$, и т. д. Стоимость каждого сокращения и расширения есть $\Theta(n)$, и количество сокращений и расширений также есть $\Theta(n)$, так что стоимость n операций есть $\Theta(n^2)$, а средняя стоимость в расчёте на одну операцию оказывается равной $\Theta(n)$.

Причина неудачи понятна: истратив весь резерв на оплату расширения таблицы, мы тут же вынуждены ее сокращать, не успев накопить средства на оплату будущего сокращения. Та же история и с расширением таблицы: мы вынуждены предпринимать его сразу после сокращения, не успев накопить средства за счет добавления записей в таблицу постоянного размера.

Дела пойдут лучше, если мы разрешим коэффициенту заполнения опускаться ниже 1/2 [введя гистерезис в наш алгоритм, как сказали бы физики]. Именно, мы по-прежнему удваиваем размер таблицы при попытке добавить запись к заполненной таблице, а вот сокращение таблицы вдвое мы предпринимаем только тогда, когда коэффициент заполнения падает ниже 1/4. Таким образом, коэффициент заполнения всегда будет больше или равен 1/4. Другими словами, мы считаем 50% оптимальным коэффициентом заполнения таблицы, и возвращаемся к такой ситуации, как только коэффициент заполнения отклонится от оптимального в два раза (в любую сторону).

Мы не будем выписывать псевдокод для алгоритма TABLE-DELETE — он аналогичен коду для TABLE-INSERT. Отметим только, что при удалении из таблицы последней записи разумно полностью освобождать память, занимаемую ячейками таблицы. Ниже мы будем считать, что в алгоритме TABLE-DELETE это предусмотрено, иными словами, что $size[T] = 0$, как только $num[T] = 0$.

Для оценки стоимости последовательности из n операций TABLE-INSERT и TABLE-DELETE, примененных к пустой таблице, мы воспользуемся методом потенциалов. Потенциальная функция будет равна нулю, когда коэффициент заполнения оптимален (равен 1/2) — так будет сразу после сокращения или расширения. Потенциал возрастает по мере того, как коэффициент заполнения приближается к 1 или 1/4. Напомним, что коэффициент заполнения $\alpha(T)$ равен $num[T]/size[T]$, если $size[T] \neq 0$, и равен 1, если $size[T]$ (и тем самым $num[T]$) равно нулю. Нашим требованиям удовлетворяет такая потенциальная функция:

$$\Phi(T) = \begin{cases} 2 \cdot num[T] - size[T], & \text{если } \alpha[T] \leqslant 1/2, \\ size[T]/2 - num[T], & \text{если } \alpha[T] \geqslant 1/2. \end{cases} \tag{18.5}$$

Заметим, что $\Phi(T) = 0$ при $\alpha(T) = 1/2$ (по любой из двух формул), что потенциал пустой таблицы равен нулю и что потенциал всегда неотрицателен. Если коэффициент заполнения равен 1 или 1/4, то $\Phi[T] = num[T]$, так что накопленного потенциала достаточно для оплаты расширения или сокращения таблицы. На рис. 18.4 изображено, как меняется потенциал в процессе выполнения последовательности операций TABLE-INSERT и TABLE-DELETE.

Найдем учётные стоимости операций при таком потенциале Φ. Если операция TABLE-INSERT или TABLE-DELETE не сопровождается расширением или сокращением таблицы, то изменение потенциала не превосходит 2 ($size$ не меняется, num меняется на 1), так что учётная стоимость не превосходит 3. В случае расширения или сжатия таблицы учётная стоимость оказывается равной 1, так как изменения в потенциале компенсируют затраты на копирование.

Рисунок 18.4 Размер таблицы ($size_i$), число записей (num_i) и потенциал

$$\Phi_i = \begin{cases} 2 \cdot num_i - size_i, & \text{если } \alpha_i \geqslant 1/2, \\ size_i/2 - num_i, & \text{если } \alpha_i < 1/2, \end{cases}$$

как функция от числа операций Table-Insert и Table-Delete, обозначенного буквой i. График num — сплошная тонкая линия, график $size$ — сплошная жирная линия, график Φ — пунктир. Непосредственно перед расширением или сокращением таблицы потенциал равен количеству записей.

В общем и целом, учётная стоимость каждой из операций есть $O(1)$, так что фактическая стоимость последовательности из n операций есть $O(n)$.

Упражнения

18.4-1 Объясните, почему для случая $\alpha_{i-1} < \alpha_i \leqslant 1/2$ учётная стоимость операции Table-Insert равна нулю.

18.4-2 Почему динамическую хеш-таблицу с открытой адресацией разумно считать заполненной (и расширять) ещё до того, как коэффициент заполнения станет равным единице? Как запрограммировать добавление записи в такую таблицу, чтобы математическое ожидание учётной стоимости добавления была $O(1)$? Почему нельзя гарантировать, что математическое ожидание реальной стоимости каждого добавления будет $O(1)$?

18.4-3 Докажите, что учётная стоимость (относительно потенциала (18.5)) операции Table-Delete, примененной к динамической таблице с коэффициентом заполнения $\geqslant 1/2$, не превосходит некоторой константы.

18.4-4 Рассмотрим такую реализацию операции Table-Delete: если в результате удаления элемента коэффициент заполнения падает ниже 1/3, то мы сокращаем таблицу на треть её размера. Пользуясь потенциалом

$$\Phi(T) = |2 \cdot \mathit{num}[T] - \mathit{size}[T]|,$$

покажите, что при этом учётная стоимость операции Table-Delete ограничена.

Задачи

18-1 Двоичный счетчик с обратным порядком битов

В главе 32 мы расскажем о важном алгоритме, называемом быстрым преобразованием Фурье. Этот алгоритм начинается с **обращения битов индекса** (bit-reversal permutation) у массива $A[0\,..\,n-1]$, где $n = 2^k$ (k — натуральное число). Именно, каждое $A[a]$ заменяется на $A[\mathrm{rev}_k(a)]$, где $\mathrm{rev}_k(a)$ получается, если представить число a в виде последовательности k битов, а затем написать эти биты в обратном порядке. Иными словами, если $a = \sum_{i=0}^{k-1} a_i 2^i$, то

$$\mathrm{rev}_k(a) = \sum_{i=0}^{k-1} a_{k-i-1} 2^i.$$

Пример: $\mathrm{rev}_4(3) = 12$, поскольку 3 в двоичной записи есть 0011, а 1100 — двоичная запись числа 12.

а. Легко вычислить $\mathrm{rev}_k(a)$ за время $\Theta(k)$. Как произвести операцию обращения битов индекса в массиве длиной $n = 2^k$ за время $O(nk)$?

Чтобы выполнять обращение битов индекса быстрее, можно применить амортизационный анализ к процедуре прибавления единицы наоборот, которая даёт $\mathrm{rev}_k(a+1)$ по данному $\mathrm{rev}_k(a)$. Выделим массив битов длиной k для хранения счетчика с обращённым порядком битов. Нам нужно разработать процедуру Bit-Reversed-Increment, переводящую $\mathrm{rev}_k(a)$ в $\mathrm{rev}_k(a)+1$. Если, например, $k = 4$ и первоначально в счетчике с обращённым порядком битов было записано 0000, то в результате последовательных вызовов процедуры Bit-Reversed-Increment значения счётчика будут равны

$$0000, 1000, 0100, 1100, 0010, 1010, \ldots = 0, 8, 4, 12, 2, 10, \ldots .$$

б. Пусть компьютер умеет за единичное время проводить с k-битными словами такие операции, как сдвиг на произвольное количество битов, логические И и ИЛИ и т.д. Как нужно реализовать процедуру Bit-Reversed-Increment, чтобы обращение битов индекса у массива длиной $n = 2^k$ занимало $O(n)$ времени?

в. Пусть за единичное время k-битное слово можно сдвинуть лишь на один бит. Можно ли в этом случае провести обращение битов индекса за время $O(n)$?

18-2 Динамический двоичный поиск

Очень путано!

Чтобы сократить время на добавление элемента к отсортированному массиву, можно поступить так. Распределим элементы отсортированного массива длины n по $k = \lceil \log_2 n \rceil$ массивам $A_0, A_1, \ldots, A_k$ следующим образом: если $\langle n_{k-1}, n_{k-2}, \ldots, n_0 \rangle$ — двоичное представление числа n (т.е. $n = \sum_{i=0}^{k-1} n_i 2^i$), то массив A_i пуст, если $n_i = 0$, и содержит 2^i элементов, если $n_i = 1$; в этом последнем случае массив A_i отсортирован. На распределение n элементов по массивам $A_0, \ldots, A_n$ никаких условий не накладывается.

а. Реализуйте для этой структуры данных операцию Search (искать). Каково ее время работы в худшем случае?

б. Реализуйте операцию Insert (добавить элемент). Оцените ее стоимость в худшем случае и учётную стоимость (если возможны только добавления, но не удаления).

в. Как реализовать операцию Delete (удалить)?

18-3 Сбалансированные по весу деревья

Пусть x — узел двоичного дерева; через $size[x]$ обозначим число листьев в поддереве с вершиной в x. Пусть число α удовлетворяет неравенству $1/2 \leqslant \alpha < 1$. Будем говорить, что узел x **α-сбалансирован** (α-balanced), если

$$size[left[x]] \leqslant \alpha \cdot size[x]$$

и

$$size[right[x]] \leqslant \alpha \cdot size[x].$$

Двоичное дерево называется **α-сбалансированным**, если все его узлы, кроме листьев, α-сбалансированы. Подход к сбалансированным деревьям, о котором идет речь ниже, был предложен Г. Варгезе (G. Varghese).

а. Пусть x — узел двоичного дерева поиска. Объясните, как перестроить поддерево с корнем в x, чтобы оно стало 1/2-сбалансированным (самое сильное требование сбалансированности) Ваш алгоритм должен работать за время $\Theta(size[x])$ и пользоваться дополнительной памятью объема $O(size[x])$.

б. Покажите, что поиск в α-сбалансированном двоичном дереве с n узлами требует в худшем случае времени $O(\log n)$.

Далее будем считать, что $\alpha > 1/2$. Будем считать, что после выполнения операций Insert и Delete (реализованных стандартным образом), производится балансировка: если какой-то узел дерева перестал быть α-сбалансированным, выбирается ближайший к корню из таких узлов и соответствующее поддерево перестраивается в 1/2-сбалансированное (см. пункт а).

Проанализируем эту схему балансировки с помощью метода потенциалов. Как обычно, потенциал будет тем больше, чем дальше дерево от сбалансированного. Пусть x — узел в двоичном дереве T; положим

$$\Delta(x) = |size[left[x]] - size[right[x]]|$$

и определим потенциал $\Phi(T)$ по формуле

$$\Phi(T) = c \sum_{x \in T : \Delta(x) \geqslant 2} \Delta(x),$$

где c — достаточно большая положительная константа, зависящая от α.

в. Покажите, что потенциал двоичного дерева всегда неотрицателен и что потенциал 1/2-сбалансированного дерева равен нулю.

г. Будем считать, что стоимость перестройки дерева с m вершинами в 1/2-сбалансированное равна m (напомним, что число операций есть $O(m)$). Какова должна быть величина c для данного α, чтобы учётная стоимость перестройки поддерева, не являющегося α-сбалансированным, была $O(1)$?

д. Покажите, что учётная стоимость удаления или вставки элемента в α-сбалансированное дерево с n вершинами есть $O(\log n)$.

Замечания

Метод группировки в амортизационном анализе описан у Ахо, Хопкрофта и Ульмана [4]. Тарьян [189] рассматривает методы предоплаты и потенциалов и приводит некоторые приложения. Согласно Тарьяну, метод предоплаты восходит к работам многих авторов, в том числе Брауна (M. R. Brown), Тарьяна (R. E. Tarjan), Хадлстона (S. Huddleston) и Мельхорна (K. Mehlhorn), в то время как метод потенциалов изобретен Слеатором (D. D. Sleator). Термин "*амортизационный анализ*" введён Слеатором и Тарьяном.

V Более сложные структуры данных

Введение

Как и в части III, в этой части мы рассмотрим структуры данных, которые хранят изменяющиеся множества — но их анализ будет несколько более сложным. В частности, в двух главах мы применяем технику амортизационного анализа (глава 18).

В главе 19 мы рассматриваем Б-деревья — сбалансированные деревья определённого вида, удобные для хранения информации на дисках. Специфика дисков в том, что важно не столько время вычислений, сколько число операций чтения/записи блоков. Число таких операций пропорционально высоте дерева, и потому высоту Б-деревьев важно поддерживать небольшой.

В главах 20 и 21 мы рассматриваем различные реализации сливаемых куч — структуры данных, которая поддерживает операции добавления элемента (Insert), отыскания минимума (Minimum), удаления минимального элемента (Extract-Min) и объединения (Union) двух куч. Помимо этих операций, могут быть эффективно реализованы также операции удаления элемента и уменьшения его ключа.

Биномиальные кучи (глава 20) выполняют каждую операцию (в худшем случае) за время $O(\lg n)$, где n — число элементов в куче (или в двух сливаемых кучах). Их преимущество (по сравнению с двоичными кучами) состоит в возможности быстрого слияния двух куч (для двоичных куч это требует времени $\Theta(n)$).

Фибоначчиевы кучи (глава 21) ещё более эффективны (по крайней мере теоретически: имеют лучшую асимптотику). Правда, здесь речь идёт уже об учётной стоимости операций. Операции Insert, Minimum и Union имеёт учётную (а также фактическую) стоимость $O(1)$. Операции Extract-Min и Delete имеют учётную стоимость $O(\lg n)$. Наиболее существенна возможность быстро выполнять операцию Decrease-Key (её учётная стоимость есть $O(1)$). Именно благодаря этому многие (на сегодняшний день) *"рекордно"* быстрые алгоритмы обработки графов используют фибоначчиевы кучи.

Наконец, в главе 22 мы рассматриваем структуры данных для хранения непересекающихся множеств (отношений эквивалентности). Мы имеем в виду следующее: имеется некоторое конечное множество, разбитое на классы. В начальный момент каждый класс содержит по одному элементу; затем их можно попарно объединять. В любой момент один из элементов класса считается его представителем; операция FIND-SET(x) даёт представитель класса, содержащего x, операция UNION(x, y) объединяет два класса. Оказывается, что весьма простое представление этой информации в виде корневого дерева весьма эффективно: последовательность из m операций требует времени $O(m\alpha(m,n))$, где $\alpha(m,n)$ — исключительно медленно растущая функция. Правда, доказать эту оценку весьма непросто (несмотря на простоту самой структуры данных), и мы ограничимся доказательство чуть менее сильной оценки.

Разумеется, этот раздел книги никак не претендует на полноту — множество интересных структур данных в него не вошли. Укажем некоторые из них:

- Структура данных, поддерживающая операции отыскания минимума, максимума, добавления и удаления элемента, поиска, удаления минимального и максимального элементов, поиск предыдущего и следующего элементов за время $O(\lg\lg n)$ в худшем случае — в предположении, что все ключи являются целыми числами от 1 до n (ван Эмде Боас [194]).
- ДИНАМИЧЕСКИЕ ДЕРЕВЬЯ (dynamic trees), которые предложили Слеатор и Тарьян [177] (см. также Тарьян [188]). Эта структура данных хранит лес из непересекающихся корневых деревьев. Каждое ребро каждого дерева имеет некоторый вещественную стоимость. Можно искать родителей, корни, стоимости рёбер, а также минимальную стоимость ребра на пути от данной вершины к корню. Можно удалять рёбра, менять стоимости рёбер на пути к корню, прививать корень дерева к другому дереву, а также делать заданную вершину корнем дерева, в котором она находится. Все эти операции можно реализовать с учётной стоимостью $O(\lg n)$; более сложная реализация гарантирует время работы $O(\lg n)$ и в худшем случае.
- РАСШИРЯЮЩИЕСЯ ДЕРЕВЬЯ (splay trees) также предложили Слеатор и Тарьян [178] (см. также Тарьян [188]). Они представляют собой двоичные деревья с обычными для деревьев поиска операциями; их учётная стоимость составляет $O(\lg n)$ (за счёт того, что время от времени дерево подвергается балансировке). Расширяющиеся деревья можно применить при реализации динамических деревьев.
- *Структуры данных с сохранением предыдущих версий* (persistent

data structures) позволяют получать информацию о предыдущих состояниях (версиях содержимого) структуры, а иногда и менять предыдущие версии. Общую методику сохранения предыдущих версий списочной структуры данных (с небольшими потерями по времени и памяти) предложили Дрисколл, Сарнак, Слеатор и Тарьян [59]. Напомним, что в задаче 14-1 подобная структура строилась для динамического множества.

19

Б-деревья

В этой главе мы рассмотрим один из видов сбалансированных деревьев, при котором обеспечивается эффективное хранение информации на магнитных дисках и других устройствах с прямым доступом — Б-деревья.

Б-деревья похожи на красно-черные деревья; разница в том, что в Б-дереве вершина может иметь много детей, на практике до тысячи (в зависимости от характеристик используемого диска). Благодаря этому константа в оценке $O(\log n)$ для высоты дерева существенно меньше, чем для черно-красных деревьев.

Как и черно-красные деревья, Б-деревья позволяют реализовать многие операции с множествами размера n за время $O(\log n)$.

Пример Б-дерева показан на рис. 19.1. Вершина x, хранящая $n[x]$ элементов, имеет $n[x]+1$ детей. Хранящиеся в x ключи служат границами, разделяющими всех её потомков на $n[x]+1$ групп; за каждую группу отвечает один из детей x. При поиске в Б-дереве мы сравниваем искомый ключ с $n[x]$ ключами, хранящимися в x, и по результатам сравнения выбираем один из $n[x]+1$ путей.

Точное определение Б-дерева и логарифмическая (от числа вершин) оценка его высоты приводятся в разделе 19.1. В разделе 19.2 описаны операции поиска и добавления элемента в дерево; удаление обсуждается в 19.3. Но прежде надо объяснить, в чем отличие магнитных дисков от оперативной памяти, делающее выгодным использование Б-деревьев.

Рисунок 19.1 Б-дерево с английскими согласными в качестве ключей. Внутренняя вершина x с $n[x]$ ключами имеет $n[x]+1$ детей. Все листья имеют одну и ту же глубину. Светлые вершины просмотрены в процессе поиска буквы R.

Рисунок 19.2 Дисковод

Структуры данных на диске

Компьютер использует разные виды памяти. **Оперативная память** (main memory) представляет собой микросхемы, каждая из которых вмещает миллионы битов. Цена такой памяти (в расчёте на бит) выше, чем для вторичной памяти (secondary storage). Типичный компьютер использует в качестве вторичной памяти **жёсткий диск**, объём которого может на несколько порядков превосходить объём оперативной памяти.

Магнитная головка (см. рис. 19.2) записывает и читает данные на магнитном слое вращающегося диска. Рычаг может переместить её вдоль радиуса диска. После этого головка читает/пишет данные на одной из **дорожек** (tracks) диска. Обычно каждая дорожка делится на определённое число равных по размеру **секторов** (каждый может занимать несколько килобайтов). Обычно записывают или считывают сектор целиком. **Время доступа** (access time), которое требуется чтобы подвести головку к нужному месту диска, может быть достаточно большим (до 20 миллисекунд); в оперативной памяти такой задержки нет, потому что нет механического перемещения. Как только головка диска уже установлена, запись или чтение диска происходит довольно быстро. Часто получается, что обработка прочитанного занимает меньше времени, чем поиск нужного сектора. Поэтому, оценивая качество алгоритма мы будем учитывать два параметра:

- число обращений к диску, и
- время вычислений (время работы процессора).

Строго говоря, время доступа зависит от положения головки относительно нужного сектора. Но мы не обращаем на это внимания и учитываем лишь число обращений к диску (т.е. число секторов, которые нужно считать или записать).

Алгоритмы, работающие с Б-деревьями, хранят в оперативной

Рисунок 19.3 Б-дерево высоты 2 содержит более миллиарда ключей. Каждая вершина содержит 1000 ключей. Всего имеется 1001 вершина на глубине 1 и более миллиона листьев на глубине 2. В каждой вершине x записано число $n[x]$ ключей в ней.

памяти лишь небольшую часть всей информации (фиксированное число секторов), и поэтому её размер не ограничивается размером доступной памяти.

Мы рассматриваем диск как большой участок памяти, работа с которым происходит следующим образом: перед тем как работать с объектом x, мы должны выполнить специальную операцию Disk-Read(x) (чтение с диска). После внесения изменений в наш объект x, мы выполняем операцию Disk-Write(x) (запись на диск):

```
...
$x \leftarrow$ указатель на какой-то объект
Disk-Read($x$)
операции, которые используют и/или изменяют
        поля объекта, на который указывает $x$.
Disk-Write($x$)      ▷ Не нужно, если
                     ▷ поля объекта не изменились.
операции, которые пользуются полями указателя $x$,
        но не меняют их.
...
```

Время работы программы в основном определяется количеством операций Disk-Read и Disk-Write, так что имеет смысл читать/записывать возможно больше информации за раз и сделать так, чтобы вершина Б-дерева заполняла полностью один сектор диска. Таким образом, степень ветвления (число детей вершины) определяется размером сектора.

Типичная степень ветвления Б-деревьев находится между 50 и 2000 (в зависимости от размера элемента). Увеличение степени ветвления резко сокращает высоту дерева, и тем самым число обращений к диску, при поиске. На рис. 19.3 показано Б-дерево степени 1001 и высоты 2, хранящее более миллиарда ключей. Учитывая, что корень можно постоянно хранить в оперативной памяти, достаточно *двух* обращений к диску, при поиске нужного ключа!

19.1 Определение Б-дерева

Как и раньше, для простоты мы считаем, что дополнительная информация, связанная с ключом, хранится в той же вершине дерева. (На практике это не всегда удобно, и в реальном алгоритме вершина может содержать лишь ссылку на сектор, где хранится эта дополнительная информация.) Мы считаем, что при перемещениях ключа дополнительная информация (или ссылка на неё) перемещается вместе с ним. Тем самым элементом Б-дерева будет ключ вместе со связанной с ним информацией.

В принципе мы могли бы использовать другую часто встречающуюся организацию Б-деревьев: помещать сопутствующую информацию в листьях (где больше места, так как не надо хранить ключи), а во внутренних вершинах хранить только ключи и указатели на детей, экономя место (и получая возможность увеличить степень ветвления при том же размере сектора).

Итак, **Б-деревом** (B-tree) назовём корневое дерево, устроенное следующим образом:

1. Каждая вершина x содержит поля, в которых хранятся:

   количество $n[x]$ ключей, хранящихся в ней;

   сами ключи $key_1[x] \leqslant key_2[x] \leqslant \ldots \leqslant key_{n[x]}[x]$ (в неубывающем порядке)

   булевское значение $leaf[x]$, истинное, когда вершина является листом.

2. Если — внутренняя вершина, то она также содержит $n[x]+1$ указатель $c_1[x], c_2[x], \ldots, c_{n[x]+1}[x]$ на её детей. У листьев детей нет, и эти поля для них не определены.

3. Ключи $key_i[x]$ служат границами, разделяющими значения ключей в поддеревьях:

   $$k_1 \leqslant key_1[x] \leqslant k_2 \leqslant key_2[x] \leqslant \cdots \leqslant key_{n[x]}[x] \leqslant k_{n[x]+1},$$

   где k_i — любой из ключей, хранящихся в поддереве с корнем $_i[x]$.

4. Все листья находятся на одной и той же глубине (равной высоте h дерева).

5. Число ключей, хранящихся в одной вершине, ограничено сверху и снизу; границы задаются единым для всего дерева числом $t \geqslant 2$, которое называется **минимальной степенью** (minimum degree) Б-дерева. Именно:

   Каждая вершина, кроме корня, содержит по меньшей мере $t-1$ ключ. Таким образом, внутренние вершины (кроме корня) имеют не менее t детей. Если дерево непусто, то в корне должен храниться хотя бы один ключ.

В вершине хранится не более $2t-1$ ключей. Следовательно, внутренняя вершина имеет не более $2t$ детей. Вершину, хранящую ровно $2t-1$ ключей, назовем **полной** (full).

В простейшем случае $t=2$, тогда у каждой внутренней вершины 2, 3 или 4 ребенка, и мы получаем так называемое **2-3-4 дерево** (2-3-4 tree). (Как мы уже говорили, для эффективной работы с диском на практике t надо брать гораздо бо́льшим.)

Высота Б-дерева

Число обращений к диску для большинства операций пропорционально высоте Б-дерева. Оценим сверху эту высоту.

Теорема 19.1. *Для всякого Б-дерева высоты h и минимальной степени $t \geqslant 2$, хранящего $n \geqslant 1$ ключей, выполнено неравенство*

$$h \leqslant \log_t \frac{n+1}{2}.$$

Доказательство. Пусть высота Б-дерева равна заданному числу h. Наименьшее число вершин в дереве будет, если степень каждой вершины минимальна, то есть у корня 2 ребёнка, а у внутренних вершин по t детей. В этом случае на глубине 1 мы имеем 2 вершины, на глубине 2 имеем $2t$ вершин, на глубине 3 имеем $2t^2$ вершин, ..., на глубине h имеем $2t^{h-1}$ вершин. При этом в корне хранится один ключ, а во всех остальных вершинах по $t-1$ ключей. (На рис. 19.4 показано такое дерево при $h=3$.) Таким образом, получаем неравенство:

$$n \geqslant 1+(t-1)\sum_{i=1}^{h} 2t^{i-1} = 1+2(t-1)\left(\frac{t^h-1}{t-1}\right) = 2t^h-1,$$

откуда следует утверждение теоремы. □

Как и для красно-чёрных деревьев, высота Б-дерева с n вершинами есть $O(\log n)$, но основание логарифма для Б-деревьев гораздо больше, что примерно в $\lg t$ раз сокращает количество обращений к диску.

Упражнения

19.1-1 Почему в определении Б-дерева требование $t \geqslant 2$ существенно?

19.1-2 При каких t дерево на рис. 19.1 можно рассматривать как Б-дерево минимальной степени t?

Рисунок 19.4 Б-дерево высоты 3 содержит минимально возможное число ключей. Внутри каждой вершины x записано число $n[x]$ ключей в ней.

19.1-3 Найти все Б-деревья минимальной степени 2, представляющие множество $\{1, 2, 3, 4, 5\}$

19.1-4 Найти точную верхнюю оценку для числа ключей, хранящихся в Б-дереве высоты h и минимальной степени t.

19.1-5 Описать структуру данных, которая получится, если в красно-чёрном дереве каждую чёрную вершину соединить с ее красными детьми, а их детей сделать детьми этой чёрной вершины.

19.2 Основные операции с Б-деревьями

В этом разделе мы подробно рассмотрим операции B-Tree-Search (поиск), B-Tree-Create (создание Б-дерева) и B-Tree-Insert (добавление элемента). Мы считаем, что:

- Корень Б-дерева всегда находится в оперативной памяти, т.е. операция Disk-Read для корня никогда не требуется; однако всякий раз, когда мы изменяем корень, мы должны его сохранять на диске.
- Все вершины, передаваемые как параметры, уже считаны с диска.

Наши процедуры обрабатывают дерево за один проход от корня к листьям.

Поиск в Б-дереве

Поиск в Б-дереве похож на поиск в двоичном дереве. Разница в том, что в каждой вершине x мы выбираем один вариант из $(n[x]+1)$, а не из двух.

Как и процедура Tree-Search (для двоичных деревьев), рекурсивной процедура B-Tree-Search получает на вход указатель x на корень поддерева и ключ k, который мы ищем в этом поддереве. Для поиска ключа k в Б-дереве T следует скомандовать B-Tree-Search($root(T), k$), где $root(T)$ указывает на корень. Если процедура обнаруживает ключ k в дереве, она возвращает пару (y, i), где y — вершина, i — порядковый номер указателя, для которого $key_i[y] = k$. В противном случае процедура возвращает константу NIL.

```
B-Tree-Search($x, k$)
1  $i \leftarrow 1$
2  while $i \leqslant n[x]$ and $k > key_i[x]$
3      do $i \leftarrow i + 1$
4  if $i \leqslant n[x]$ and $k = key_i[x]$
5     then return $(x, i)$
6  if $leaf[x]$
7     then return NIL
8     else Disk-Read($c_i[x]$)
9          return B-Tree-Search($c_i[x], k$)
```

В строках 1-3 ищется наименьшее i, для которого $k \leqslant key_i[x]$; если такого нет, то в i помещается $n[x]+1$ (линейный поиск). Если нужный ключ найден, работа прекращается (строки 4–5). Затем (строки 6–7) программа либо останавливается, если поиск завершился безрезультатно (x — лист), либо рекурсивно вызывает себя, предварительно считав с диска корень нужного поддерева (строки 8–9).

На рис. 19.1 показана работа этой программы. Светлые вершины просмотрены при поиске буквы R.

Как и процедура Tree-Search, наша программа просматривает вершины дерева от корня к листу. Поэтому число обращений к диску есть $\Theta(h) = \Theta(\log_t n)$, где h — высота дерева, а n — количество ключей. Так как $n[x] \leqslant 2t$, то цикл **while** в строках 2–3 повторяется $O(t)$ раз, и время вычислений равно $O(th) = O(t \log_t n)$.

Создание пустого Б-дерева

При работе с Б-деревом T, мы сначала создаем с помощью процедуры B-Tree-Create пустое дерево, а затем записываем в него данные с помощью процедуры B-Tree-Insert. Обе эти процедуры используют подпрограмму Allocate-Node, которая находит место на диске для новой вершины. Мы считаем, что подпрограмма Allocate-Node требует времени $O(1)$ и не использует операцию Disk-Read.

Рисунок 19.5 Разбиение вершины дерева минимальной степени $t = 4$. Делим вершину y на две: y и z. Ключ-медиана S вершины y переходит к ее родителю x.

```
B-TREE-CREATE($T$)
$x \leftarrow$ ALLOCATE-NODE()
$leaf[x] \leftarrow$ TRUE
$n[x] \leftarrow 0$
DISK-WRITE($x$)
$root[T] \leftarrow x$
```

Процедура B-TREE-CREATE требует одного обращения к диску, время работы — $O(1)$.

Разбиение вершины Б-дерева на две

Добавление элемента в Б-дерево — значительно более сложная операция, чем добавление элемента в двоичное дерево поиска. Ключевым местом является **разбиение** (splitting) полной (с $2t - 1$ ключами) вершины y на две вершины, имеющие по $t - 1$ элементов в каждой. При этом **ключ-медиана** (median key) $key_t[y]$ отправляется к родителю x вершины y и становится разделителем двух полученных вершин. Это возможно, если вершина x неполна. Если y — корень, процедура работает аналогично. В этом случае высота дерева увеличивается на единицу. Таков механизм роста Б-дерева.

Процедура B-TREE-SPLIT-CHILD делит вершину y на две и меняет соответствующим образом её родителя x. На рис. 19.5 показано, как это происходит. Ключ-медиана S вершины y переходит к её родителю x, а бо́льшие S элементы переписываются в нового ребенка z вершины x. Входными данными процедуры являются *неполная* внутренняя вершина x, число i и полная вершину y, для которых $y = c_i[x]$. Мы считаем, что вершины x и y уже находятся в оперативной памяти.

```
B-Tree-Split-Child$(x, i, y)$
 1  $z \leftarrow$ Allocate-Node()
 2  $leaf[z] \leftarrow leaf[y]$
 3  $n[z] \leftarrow t - 1$
 4  for $j \leftarrow 1$ to $t - 1$
 5       do $key_j[z] \leftarrow key_{j+t}[y]$
 6  if not $leaf[y]$
 7     then for $j \leftarrow 1$ to $t$
 8              do $c_j[z] \leftarrow c_{j+t}[y]$
 9  $n[y] \leftarrow t - 1$
10  for $j \leftarrow n[x] + 1$ downto $i + 1$
11       do $c_{j+1}[x] \leftarrow c_j[x]$
12  $c_{i+1}[x] \leftarrow z$
13  for $j \leftarrow n[x]$ downto $i$
14       do $key_{j+1}[x] \leftarrow key_j[x]$
15  $key_i[x] \leftarrow key_t[y]$
16  $n[x] \leftarrow n[x] + 1$
17  Disk-Write$(y)$
18  Disk-Write$(z)$
19  Disk-Write$(x)$
```

Вершина y имела $2t$ детей; после преобразования в ней осталось t наименьших из них, а остальные t стали детьми новой вершины z, которая в свою очередь стала ребёнком вершины x. Ключ-медиана вершины y добавлен к вершине x и стал разделителем между вершиной y и следующей за ней вершиной z.

Строки 1–8 формируют вершину z и передают ей детей. Строка 9 меняет вершину y. Наконец, строки 10–16 вносят соответствующие изменения в вершину x. Строки 17–19 сохраняют изменения на диске. Время работы циклов (строки 4–5 и 7–8) равно $\Theta(t)$. (Для остальных циклов требуется не больше t шагов).

Добавление элемента в Б-дерево

Процедура B-Tree-Insert добавляет элемент k в Б-дерево T, пройдя один раз от корня к листу. На это требуется время $O(th) = O(t \log_t n)$ и $O(h)$ обращений к диску, если высота дерева h. По ходу дела мы с помощью процедуры B-Tree-Split разделяем встречающиеся нам полные вершины, используя такое наблюдение: если полная вершина имеет неполного родителя, то её можно разделить, так как в родителе есть место для дополнительного ключа. В конце концов мы оказываемся в неполном листе, куда и добавляем новый элемент.

Рисунок 19.6 Рис. 19.6. Корень r Б-дерева (с $t = 4$) является полной вершиной. Он делится на две половины; при этом создается новый корень s, детьми которого становятся эти вершины. Новый корень s содержит ключ-медиану старого корня. Высота Б-дерева увеличилась на единицу.

```
B-Tree-Insert($T, k$)
 1  $r \leftarrow root[T]$
 2  if $n[r] = 2t - 1$
 3     then $s \leftarrow$ Allocate-Node()
 4          $root[T] \leftarrow s$
 5          $leaf[s] \leftarrow$ FALSE
 6          $n[s] \leftarrow 0$
 7          $c_1[s] \leftarrow r$
 8          B-Tree-Split-Child($s, 1, r$)
 9          B-Tree-Insert-Nonfull($s, k$)
10     else B-Tree-Insert-Nonfull($r, k$)
```

Строки 3–9 относятся к случаю добавления в дерево с полным корнем (пример на рис. 19.6). Именно в этом случае увеличивается высота Б-дерева. Отметим, что точкой роста Б-дерева является корень (а не лист, как в двоичных деревьях поиска).

Сделав корень неполным (если он не был таковым с самого начала), мы вызываем процедуру B-Tree-Insert-Nonfull(x, k), которая добавляет элемент k в поддерево с корнем в неполной вершине x. Эта процедура рекурсивно вызывает себя, при необходимости (если вершина оказалась полной) выполнив разделение.

```
B-Tree-Insert-Nonfull$(x, k)$
 1  $i \leftarrow n[x]$
 2  if $leaf[x]$
 3     then while $i \geqslant 1$ and $k < key_i[x]$
 4              do $key_{i+1}[x] \leftarrow key_i[x]$
 5                 $i \leftarrow i - 1$
 6           $key_{i+1}[x] \leftarrow k$
 7           $n[x] \leftarrow n[x] + 1$
 8           Disk-Write$(x)$
 9     else while $i \geqslant 1$ and $k < key_i[x]$
10              do $i \leftarrow i - 1$
11           $i \leftarrow i + 1$
12           Disk-Read$(c_i[x])$
13           if $n[c_i[x]] = 2t - 1$
14             then B-Tree-Split-Child$(x, i, c_i[x])$
15                  if $k > key_i[x]$
16                    then $i \leftarrow i + 1$
17           B-Tree-Insert-Nonfull$(c_i[x], k)$
```

Эта процедура работает следующим образом. Если вершина x — лист, то ключ k в него добавляется (строки 3–8; напомним, что вершина x предполагается неполной). В противном случае нам нужно добавить k к поддереву, корень которого является ребёнком x. В строках 9–11 мы находим нужного ребёнка вершины x. Если этот ребёнок оказывается полной вершиной (строка 13), то в строке 14 он разделяется на две неполных вершины, и в строках 15–16 определяется, в какое из новых поддеревьев следует добавить k. (Заметим, что после изменения i в строке 16 нет необходимости обращаться к диску — вновь созданная вершина, к которой мы обратимся в строке 17, уже находится в оперативной памяти.) Таким образом, мы знаем, что вершина $c_i[x]$ — неполная, и в строке 17 добавляем ключ k в соответствующее поддерево с помощью рекурсивного вызова процедуры B-Tree-Insert-Nonfull. На рис. 19.7 показаны разные случаи добавления элемента в Б-дерево.

Процедура B-Tree-Insert-Nonfull использует $O(1)$ операций Disk-Read и Disk-Write (если не считать рекурсивного вызова). Следовательно, для Б-дерева высоты h процедура B-Tree-Insert обращается к диску $O(h)$ раз. Время вычислений есть $O(th) = O(t \log_t h)$. В процедуре B-Tree-Insert-Nonfull рекурсивный вызов является последним оператором (tail recursion), и поэтому мы могли бы заменить рекурсию циклом. Отсюда видно, что число секторов, которые необходимо держать в оперативной памяти, есть $O(1)$.

Рисунок 19.7 Добавление элемента в Б-дерево, для которого $t = 3$, (вершина содержит до 5 ключей). Светлые вершины были изменены при добавлении. (а) Начальное дерево. (б) Добавили букву B (в неполный лист). (в) Добавление Q приводит к разделу вершины $RSTUV$ на RS и UV, буква T перешла в корень, после чего букву Q добавили к вершине RS. (г) В предыдущее дерево добавили L: корень был полной вершиной, поэтому его пришлось разделить на две и высота дерева увеличилась на единицу. Букву L вставили в лист JK. (д) В предыдущее дерево добавили F. Вершину $ABCDE$ разделили на две и в половинку DE добавили F.

Упражнения

19.2-1 Проследить, за добавлением в пустое Б-дерево элементов

$$F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E$$

в указанном порядке (нарисовать только состояния дерева перед разделением какой-то из вершин, а также последнее состояние)

19.2-2 Выяснить, исполняются ли лишние операции Disk-Read или Disk-Write при вызове процедуры B-Tree-Insert. (Лишняя операция Disk-Read читает сектор, уже загруженный в оперативную память. Лишняя операция Disk-Write сохраняет на диске не изменившийся сектор.)

19.2-3 Как найти минимальный элемент в Б-дереве? Как найти элемент Б-дерева, предшествующий данному элементу?

19.2-4* Ключи $1, 2, \ldots, n$ добавляют по одному в пустое Б-дерево минимальной степени 2. Сколько вершин у полученного Б-дерева?

19.2-5 Так как у листьев нет указателей на детей, то в них может поместиться больше ключей, чем во внутренние вершины. Как будут выглядеть процедуры создания Б-дерева и добавления в него элемента, использующие это обстоятельство?

19.2-6 Заменим в процедуре B-Tree-Search линейный поиск двоичным. Показать, что тогда время вычислений для этой процедуры станет равным $O(\log n)$ (константа не зависит от t!).

19.2-7 Предположим, что мы можем сами выбрать размер сектора, причем время чтения сектора, вмещающего вершину Б-дерева степени t, будет $a + bt$, где a и b — некоторые константы. Как следует выбрать t, чтобы уменьшить время поиска в Б-дереве? Оцените оптимальное значение t в случае $a = 30$ миллисекунд, $b = 40$ микросекунд.

19.3 Удаление элемента из Б-дерева

Удаление элемента из Б-дерева (процедура B-Tree-Delete) происходит аналогично добавлению, хотя немного сложнее. Мы не будем приводить процедуру удаления полностью, а объясним, как она работает.

Пусть нужно удалить ключ k из поддерева с корнем в вершине x. Наша процедура будет устроена так, что при каждом ее рекурсивном вызове вершина x содержит по меньшей мере t ключей, где t

— минимальная степень Б-дерева. По правилам вершина Б-дерева должна содержать не меньше $t-1$ ключа, так что в нашем случае имеется запасной ключ. Этот прием (следить, чтобы запасной ключ всегда был) позволяет удалить элемент, пройдя Б-дерево один раз от корня к листу и не делая шагов в обратном направлении (с единственным исключением, которое мы его разберем позже).

Договоримся, что если в результате удаления корень дерева стал пустым, то он удаляется, и его единственный ребенок становится новым корнем, при этом высота Б-дерева уменьшается на единицу, и корень уже не пуст (если только всё дерево не пусто).

На рис. 19.8 показаны разные случаи удаления элемента из Б-дерева.

1. Если ключ k находится в вершине x, являющейся листом, то удаляем k из x.
2. Если ключ k находится во внутренней вершине x, то делаем следующее:

   Если ребенок y вершины x, предшествующий k, содержит не менее t элементов, то находим ключ k', непосредственно предшествующий ключу k. Этот ключ находится в листе поддерева с корнем в y. Найти его можно за один просмотр поддерева от корня к листу. Рекурсивно вызываем процедуру: удаляем k'. Заменяем в x ключ k на k'.

   Если ребенок z, следующий за k, содержит не менее t элементов, поступаем аналогично.

   Если и y, и z содержат по $t-1$ элементу, соединяем вершину y, ключ k, вершину z, помещая всё это в вершину y, которая теперь содержит $2t-1$ ключ. Стираем z и выкидываем из x ключ k и указатель на z. Рекурсивно удаляем k из y.
3. Если x — внутренняя вершина, но ключа k в ней нет, найдем среди детей вершины x корень $c_i[x]$ поддерева, где должен лежать ключ k (если этот ключ вообще есть). Если $c_i[x]$ содержит не менее t ключей, можно рекурсивно удалить k из поддерева. Если же $c_i[x]$ содержит всего $t-1$ элемент, то предварительно сделаем шаг 3а или 3б.

   Пусть вершина $c_i[x]$ содержит $t-1$ элемент, но один из её соседей (например, правый) содержит по крайней мере t элементов. (Здесь соседом мы называем такого ребенка вершины x, который отделен от $c_i[x]$ ровно одним ключом-разделителем.) Тогда добавим ребёнку $c_i[x]$ элемент его родителя x, а родителю передадим левый элемент этого соседа. При этом самый левый ребёнок соседа станет самым правым ребенком вершины $c_i[x]$.

   Пусть оба соседа вершины $c_i[x]$ содержат по $t-1$ элементу. Тогда объединим вершину $c_i[x]$ с одним из соседей (как в случае 2в).

начальное дерево
F удалена: случай 1
M удалена: случай 2а
G удалена: случай 2б

Рисунок 19.8 Удаление элемента из Б-дерева. Для этого Б-дерева минимальная степень равна 3, т.е. вершина содержит не менее 2 элементов. Светлые вершины были изменены при удалении. (а) Б-дерево рисунка 19.7 (е). (б) Случай 1: удаление буквы F из листа. (в) Удаление буквы M. Это случай 2а: буква-предшественница L перешла на её место. (г) Удаление буквы G. Это случай 2в: сначала G отправили вниз, где образовался лист $DEGJK$, из которого G и удалили (случай 1). (д) Удаление буквы D. Это случай 3б: мы не можем рекурсивно обработать вершину CL, в которой всего два элемента, поэтому спускаем вниз P и получаем вершину $CLPTX$. После этого удаляем D из листа (случай 1). (е$'$) После удаления D корень стал пустым и мы удалили его. Высота дерева уменьшилась на единицу. (е) Удалили B. Это случай 3а: C спустили на место B, а E подняли на место C.

D удалена: случай 3б
уменьшение высоты дерева
B удалена: случай 3а

При этом ключ, разделявший их в вершине x, станет ключом-медианой новой вершины.

Описанная процедура требует более одного прохода только в случаях 2а и 2б (когда она заменяет удаляемый элемент его предшественником или последователем). Заметим, что это происходит, только если требуется удалить элемент из внутренней вершины. Большинство вершин Б-дерева — листья, так что эти случаи будут редкими.

Хотя процедура выглядит запутанно, она требует всего $O(h)$ обращений к диску для Б-дерева высоты h. (Между двумя рекурсивными вызовами выполняется $O(1)$ команд DISK-READ и DISK-WRITE). Вычисления требуют времени $O(th) = O(t \log_t h)$.

Упражнения

19.3-1 Показать результат удаления вершин C, P и V (в указанном порядке) из дерева рис. 19.8 (f).

19.3-2 Написать процедуру B-TREE-DELETE.

Задачи

19-1 Стеки на диске

Представим себе, что мы хотим реализовать стек на машине с небольшой оперативной памятью и большим жестким диском (стек не помещается в оперативную память, и должен по большей части храниться на диске).

При простейшей (но неэффективной) реализации стека на диске хранится всё, кроме переменной p (указатель стека), которая определяет место вершины стека на диске таким образом: вершиной будет $(p \bmod m)$-ый элемент $\lfloor p/m \rfloor$-го сектора диска (m — размер сектора).

Чтобы добавить элемент в стек, мы читаем соответствующий сектор, на нужное место помещаем новый элемент, увеличиваем значения указателя на единицу и снова записываем сектор на диск. Аналогично реализуется операция удаления элемента из стека. (Мы читаем сектор с диска и уменьшаем на единицу значение указателя. Так как сектор не менялся, то записывать его на диск не нужно.)

Будем учитывать количество обращений к диску, а также время вычислений, при подсчете которого каждое обращение к диску считается требующим $\Theta(m)$ единиц времени.

а. Сколько обращений к диску требуется в худшем случае для n операций со стеком при этой реализации? Чему равно общее время? (Здесь и далее требуется ответ в терминах m и n.)

Рассмотрим другую реализацию стека, при которой один сектор целиком хранится в памяти. (Кроме того, нам требуется помнить номер хранимого сектора.) По мере необходимости мы будем возвращать этот сектор на диск и считывать новый. Если нужный сектор уже находится в памяти, то обращаться к диску не нужно.

б. Сколько обращений к диску требуется для добавления n элементов в стек (в худшем случае)? Чему равно время вычислений?

в. Сколько обращений к диску требуется в худшем случае для n операций со стеком? Чему равно время вычислений?

Существует более эффективная реализация, при которой в оперативной памяти хранятся два сектора (и ещё несколько чисел).

г. Как сделать так, чтобы каждая операция требовала (при амортизационном анализе) $O(1/m)$ обращений к диску и времени $O(1)$?

19-2 Объединение и разделение 2-3-4 деревьев

Операция **объединения** (join) получает на входе два множества S' и S'' и элемент x, для которых $key[x'] < key[x] < key[x'']$ при всех $x' \in S'$ и $x'' \in S''$. Ее результатом является множество $S = S' \cup \{x\} \cup S''$. **Разделение** (split) — операция, обратная объединению. Она получает на входе множество S и элемент $x \in S$ и создаёт два других множества S' и S'', состоящих соответственно из меньших и из больших x элементов множества S. В этой задаче требуется реализовать эти операции для 2-3-4 деревьев. Для удобства будем считать, что элементы состоят только из ключей, и все ключи различны.

а. Будем хранить в каждой вершине x 2-3-4 дерева поле $height[x]$, хранящее высоту поддерева с корнем в x. Показать, что эту инфор-

хранить хранящее — это неплохо!

мацию можно поддерживать, не ухудшая асимптотику времени поиска, добавления и удаления.

б. Реализовать операцию объединения деревьев T' и T'', разделённых ключом k. Время работы должно быть $O(|h' - h''|)$, где h', h'' — высоты деревьев.

в. Пусть p — путь в 2-3-4 дереве T от корня к заданному ключу k. Рассмотрим два множества ключей из T: меньшие k (множество S') и большие k (множество S''). Показать, что S' разбивается на деревья $T'_0, T'_1, \ldots, T'_m$, разделенные ключами $k'_1, k'_2, \ldots, k'_m$ (для всех $y \in T'_{i-1}$ и $z \in T'_i$ выполнено $y < k'_i < z$ при $i = 1, 2, \ldots, m$. Как связаны высоты деревьев T'_{i-1} и T'_i? На какие части путь p делит S''?

г. Реализовать операцию разделения. Для этого следует объединить ключи из S' в 2-3-4 дерево T' и ключи из S'' в дерево T''. Для дерева с n ключами время работы этой операции должно быть $O(\log n)$. (Указание: при сложении стоимостей операций объединения происходит сокращение.)

Замечания

Сбалансированные деревья и Б-деревьев обсуждаются в Кнут [123], Ахо, Хопкрофт и Ульман [4] и Седжвик [175]. Подробный обзор Б-деревьев дан в Комер [48]. Гибас и Седжвик [93] рассмотрели взаимосвязи между разными видами сбалансированных деревьев, включая красно-чёрные и 2-3-4 деревья.

В 1970 году Хопкрофт (J. E. Hopcroft) предложил понятие 2-3 деревьев, которые явились предшественниками Б-деревьев и 2-3-4 деревьев. В этих деревьях каждая внутренняя вершина имеет 2 или 3 детей. Б-деревья были определены Байером и МакКрейтом в 1972 году [18]. В их работе не объяснён выбор названия.

20 Биномиальные кучи

В этой главе и в главе 21 рассматриваются структуры данных, известные как **сливаемые кучи** (mergeable heaps). Такая структура хранит несколько множеств (куч), элементы которых называют вершинами. Каждая вершина содержит поле *key* (ключ), в котором хранится некоторое число; кроме того, в вершине может храниться некоторая информация, сопровождающая это число. Сливаемые кучи позволяют выполнять следующие пять операций:

Make-Heap() создаёт и возвращает новую кучу, не содержащую элементов;

Insert(H, x) добавляет элемент (вершину) x в кучу H (поле *key* элемента x должно быть заполнено заранее);

Minimum(H) возвращает указатель на элемент кучи H с минимальным ключом;

Extract-Min(H) изымает элемент с минимальным ключом из кучи H и возвращает указатель на изъятый элемент;

Union(H_1, H_2) объединяет кучи H_1 и H_2, то есть создаёт и возвращает новую кучу, содержащую все элементы куч H_1 и H_2. Сами кучи H_1 и H_2 при этом исчезают.

Структуры данных, описываемые в этой и следующей главах, поддерживают ещё две операции:

Decrease-Key(H, x, k) уменьшает ключ вершины x кучи H, присваивая ему новое значение k (предполагается, что новое значение не превосходит старого);

Delete(H, x) удаляет элемент (вершину) x из кучи H.

Как видно из таблицы 20.1, если мы не нуждаемся в операции Union, то (двоичные) кучи, с помощью которых мы сортировали массив в главе 7, весьма эффективны. Для них все операции, кроме операции Union, выполняются за время $O(\lg n)$ в худшем случае (а некоторые из операций — ещё быстрее). Но для выполнения операции Union нам приходится приписывать один массив к другому и затем выполнять процедуру Heapify, что требует времени $\Theta(n)$.

В этой главе мы расскажем о *"биномиальных кучах"* (второй столбец таблицы). Обратите внимание, что объединение двух би-

Процедура	Двоичные кучи (в худшем случае)	Биномиальные кучи (в худшем случае)	Фибоначчиевы кучи (в среднем)
Make-Heap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
Minimum	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
Extract-Min	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
Union	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
Decrease-Key	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
Delete	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

Рисунок 20.1 Время выполнения различных операций для трёх видов сливаемых куч (n — общее число элементов в кучах на момент операции).

номиальных куч, содержащих в сумме n элементов, требует всего лишь $O(\lg n)$ операций.

В главе 21 мы рассматриваем *"фибоначчиевы кучи"*, которые ещё более эффективны (третий столбец). Отметим, впрочем, что это улучшение достигается лишь для учётной стоимости операций при амортизационном анализе.

В наших процедурах мы не занимаемся выделением и освобождением памяти для элементов куч.

Все три вида куч, указанных в таблице, не позволяют эффективно реализовать поиск элемента с данным ключом (Search). Поэтому процедуры Decrease-Key и Delete получают в качестве параметра не ключ вершины, а указатель на неё (во многих случаях это требование не создаёт проблем).

В разделе 20.1 определяются биномиальные деревья и кучи. Там же описывается представление биномиальных куч в программе. В разделе 20.2 показано, как реализовать все перечисленные операции за указанное в таблице 20.1 время.

20.1 Биномиальные деревья и биномиальные кучи

Биномиальная куча состоит из нескольких биномиальных деревьев.

20.1.1 Биномиальные деревья

Биномиальные деревьями (binomial trees) называются упорядоченные (в смысле раздела 5.2.2) деревья $B_0, B_1, B_2, \ldots$, определяемые индуктивно.

Дерево B_0 состоит из единственной вершины (рис. 20.2а). Дерево B_k склеено из двух экземпляров дерева B_{k-1}: корень одного из них объявлен самым левым потомком корня другого. На рис. 20.2б показаны биномиальные деревья B_0–B_4.

depth = глубина

Рисунок 20.2 (а) Рекурсивное определение биномиального дерева B_k (треугольники — поддеревья с выделенным корнем). (б) Деревья B_0–B_4. Цифры указывают глубину вершин в дереве B_4. (в) Другое представление дерева B_k.

Лемма 20.1 (Свойства биномиальных деревьев). *Дерево* B_k

1. *содержит* 2^k *вершин;*
2. *имеет высоту* k*;*
3. *имеет* C_k^i *вершин глубины* i*;*
4. *имеет корень, являющийся вершиной степени* k*; все остальные вершины имеют меньшую степень; дети корня являются корнями поддеревьев* $B_{k-1}, B_{k-2}, \ldots, B_1, B_0$ *(слева направо).*

Доказательство. проводится индукцией по k. Для B_0 всё очевидно. Пусть утверждение верно для B_{k-1}. Тогда:

1. Дерево B_k состоит из двух копий B_{k-1} и потому содержит $2^{k-1} + 2^{k-1} = 2^k$ вершин.

2. При соединении двух копий описанным способом высота увеличивается на 1.

3. Пусть $D(k, i)$ — число вершин глубины i в дереве B_k. По построению дерева B_k (см. рис. 20.2а) имеет место равенство

$$D(k, i) = D(k-1, i) + D(k-1, i-1) = C_{k-1}^i + C_{k-1}^{i-1} = C_k^i.$$

(Мы воспользовались индуктивным предположением и упр. 6.1-7.)

4. По предположению индукции все вершины в B_{k-1} имеют степень не больше $k-1$, так что корень дерева B_k будет в нём единственной вершиной степени k. В правом из склеиваемых деревьев дети корня были вершинами деревьев $B_{k-2}, B_{k-3}, \ldots, B_1, B_0$, теперь к ним добавилось слева еще дерево B_{k-1}. □

Следствие 20.2. Максимальная степень вершины в биномиальном дереве с n вершинами равна $\lg n$.

Доказательство. Используем утверждения 1 и 4 леммы 20.1. □

Название *"биномиальное дерево"* связано с утверждением 3 леммы 20.1 (число C_k^i называют биномиальным коэффициентом). См. также упр. 20.1-3.

20.1.2 Биномиальные кучи

Биномиальная куча (binomial heap) — это набор H биномиальных деревьев, в вершинах которых записаны ключи (числа) и, возможно, дополнительная информация. При этом должны быть выполнены такие **свойства биномиальной кучи** (binomial-heap properties):

1. Каждое дерево в H **упорядочено в смысле куч** (is heap-ordered), т.е. ключ каждой вершины не меньше, чем ключ её родителя.

2. В H нет двух биномиальных деревьев одного размера (с одинаковой степенью корня).

Первое свойство гарантирует, что корень каждого из деревьев имеет наименьший ключ среди его вершин.

Из второго свойства следует, что суммарное количество вершин в биномиальной куче H однозначно определяет размеры входящих в неё деревьев. В самом деле, общее число вершин, равное n, есть сумма размеров отдельных деревьев, которые суть различные степени двойки, а такое представление единственно (двоичная система счисления). Отсюда вытекает также, что куча с n элементами состоит из не более чем $\lfloor \lg n \rfloor + 1$ биномиальных деревьев.

На рис. 20.3а показана биномиальная куча H с 13 вершинами. В двоичной системе 13 записывается как 1101 ($13 = 8 + 4 + 1$), и H состоит из биномиальных деревьев B_3, B_2 и B_0 (с количеством вершин 8, 4 и 1).

Представление биномиальных куч в программе

Как показано на рис. 20.3б, каждое биномиальное дерево в биномиальной куче хранится в представлении *"левый ребёнок, правый сосед"* (см. разд. 11.4). Каждая вершина имеет поле *key* (ключ), а

Рисунок 20.3 Биномиальная куча H с $n = 13$ вершинами. (а) Куча состоит из биномиальных деревьев B_0, B_2 и B_3, имеющих соответственно 1, 4 и 8 вершин (всего 13 вершин). Ключ каждой вершины не меньше, чем ключ её родителя. Корни деревьев связаны в корневой список в порядке возрастания степени. (б) Представление кучи в программе. Каждое биномиальное дерево хранится в представлении *"левый ребёнок, правый сосед"*, и в каждой вершине хранится её степень (*degree*).

также хранит дополнительную информацию. Кроме того, каждая вершина x хранит указатели $p[x]$ (родитель), $child[x]$ (самый левый из детей) и $sibling[x]$ (*"следующий по старшинству брат"*). Если x — корень, то $p[x] = \text{NIL}$; если x не имеет детей, то $child[x] = \text{NIL}$, а если x является самым правым ребёнком своего родителя, то $sibling[x] = \text{NIL}$. Каждая вершина x содержит также поле $degree[x]$ в котором хранится степень (число детей) вершины x.

На рис. 20.3 показано также, что корни биномиальных деревьев, составляющих биномиальную кучу, связаны в список, называемый **корневым списком** (root list) в порядке возрастания степеней. Как мы видели, в куче с n вершинами степени корневых вершин образуют подмножество множества $\{0, 1, \ldots, \lfloor \lg n \rfloor\}$.

Для построения корневого списка используется поле *sibling*: для корневой вершины оно указывает на следующий элемент корневого списка (и содержит NIL для последнего корня в корневом списке).

Рисунок 20.4 Биномиальное дерево B_4 с вершинами, пронумерованными в порядке *"левое поддерево, ... , правое поддерево, вершина"*; номера записаны в двоичной системе.

Доступ к биномиальной куче H осуществляется с помощью поля $head[H]$ — указателя на первый корень в корневом списке кучи H. Если куча H пуста, то $head[H] = \text{NIL}$.

Упражнения

20.1-1 Пусть x — вершина одного из биномиальных деревьев в биномиальной куче, причём $sibling[x] \neq \text{NIL}$. Как соотносятся значения $degree[sibling[x]]$ и $degree[x]$? (Ответ зависит от того, является ли x корнем.)

20.1-2 Пусть x — некорневая вершина биномиального дерева. Сравнить $degree[p[x]]$ и $degree[x]$.

20.1-3 Расположим вершины биномиального дерева B_k в таком порядке, чтобы каждая вершина следовала за своими потомками. При этом сначала идут потомки её левого ребёнка, затем сам этот ребенок, затем потомки следующего ребенка, он сам и т.д. (postorder walk). Пронумеруем вершины, записав номера в двоичной системе счисления (рис. 20.4). Покажите, что глубина вершины определяется числом единиц (корень имеет одни единицы, его дети на одну единицу меньше и т.д.), а степень вершины равняется числу единиц у правого края двоичной записи. Сколько имеется двоичных строк длины k, содержащих ровно j единиц, и где находятся соответствующие им вершины?

20.2 Операции с биномиальными кучами

В этом разделе приведены реализации операций с биномиальными кучами. Время работы этих операций указано в таблице 20.1. Мы докажем только верхние оценки, оставляя нижние в качестве упр. 20.2-10.

Создание новой кучи

Процедура Make-Binomial-Heap создаёт и возвращает объект H, для которого $head[H] = \text{NIL}$ (время работы $\Theta(1)$).

Поиск минимального ключа

Процедура Binomial-Heap-Minimum возвращает указатель на вершину с минимальным ключом в биномиальной куче H, состоящей из n вершин. Мы используем специальное значение ∞, которое больше всех значений ключей (см. упр. 20.2-5).

```
Binomial-Heap-Minimum($H$)
y ← NIL
x ← head[H]
min ← ∞
while x ≠ NIL
    do if key[x] < min
          then min ← key[x]
               y ← x
       x ← sibling[x]
return y
```

В биномиальных деревьях наименьшие элементы стоят в корнях, так что достаточно выбрать минимальный элемент корневого списка. Процедура Binomial-Heap-Minimum просматривает этот список, храня в переменной *min* минимальное из просмотренных значений, а в переменной y — вершину, где оно достигается.

Например, для кучи рис. 20.3 процедура Binomial-Heap-Minimum возвращает указатель на вершину с ключом 1.

Длина корневого списка не превосходит $\lfloor \lg n \rfloor + 1$, поэтому время работы процедуры Binomial-Heap-Minimum есть $O(\lg n)$.

20.2.1 Объединение двух куч

Операция Binomial-Heap-Union, соединяющая две биномиальные кучи в одну, используется в качестве подпрограммы большинством остальных операций.

[Идея проста: пусть есть две биномиальные кучи с m и n элементами. Размеры деревьев в них соответствуют слагаемым в разложениях чисел m и n в сумму степеней двойки. При сложении столбиком в двоичной системе происходят переносы, которые соответствуют слияниям двух биномиальных деревьев B_{k-1} в дерево B_k. Надо только посмотреть, в каком из сливаемых деревьев корень меньше, и считать его верхним.]

Опишем операцию объединения подробно. Начнём со вспомогательной операции Binomial-Link, которая соединяет два биномиальных дерева одного размера (B_{k-1}), корнями которых являются вершины y и z, делая вершину z родителем вершины y и корнем дерева B_k.

```
BINOMIAL-LINK($y, z$)
$p[y] \gets z$
$sibling[y] \gets child[z]$
$child[z] \gets y$
$degree[z] \gets degree[z] + 1$
```

Время работы этой процедуры — $O(1)$ (удачным образом оказывается, что вершину y надо добавить в *начало* списка детей вершины z, что легко сделать в представлении *"левый ребёнок, правый сосед"*).

Теперь напишем процедуру BINOMIAL-HEAP-UNION, которая объединяет биномиальные кучи H_1 и H_2 (сами кучи при этом исчезают). Помимо процедуры BINOMIAL-LINK, нам понадобится процедура BINOMIAL-HEAP-MERGE, которая сливает корневые списки куч H_1 и H_2 в единый список, вершины в котором идут в порядке возрастания степеней. (Эта процедура аналогична процедуре MERGE из раздела 1.3.1, и её мы оставляем читателю в качестве упр. 20.2-2.)

```
BINOMIAL-HEAP-UNION($H_1, H_2$)
$H \gets$ MAKE-BINOMIAL-HEAP()
$head[H] \gets$ BINOMIAL-HEAP-MERGE($H_1, H_2$)
освободить память, занятую под $H_1$ и $H_2$
          (сохранив списки, на которые указывают $H_1$ и $H_2$)
if $head[H] =$ NIL
   then return $H$
$prev\text{-}x \gets$ NIL
$x \gets head[H]$
$next\text{-}x \gets sibling[x]$
while $next\text{-}x \neq$ NIL
     do if ($degree[x] \neq degree[next\text{-}x]$) or
                 ($sibling[next\text{-}x] \neq$ NIL
                 and $degree[sibling[next\text{-}x]] = degree[x]$)
           then $prev\text{-}x \gets x$                    ▷ Случаи 1 и 2
                $x \gets next\text{-}x$                     ▷ Случаи 1 и 2
           else if $key[x] \leqslant key[next\text{-}x]$
                   then $sibling[x] \gets sibling[next\text{-}x]$    ▷ Случай 3
                        BINOMIAL-LINK($next\text{-}x, x$)    ▷ Случай 3
                   else if $prev\text{-}x =$ NIL                ▷ Случай 4
                           then $head[H] \gets next\text{-}x$      ▷ Случай 4
                           else $sibling[prev\text{-}x] \gets next\text{-}x$▷ Случай 4
                        BINOMIAL-LINK($x, next\text{-}x$)    ▷ Случай 4
                        $x \gets next\text{-}x$                ▷ Случай 4
        $next\text{-}x \gets sibling[x]$
return $H$
```

На рисунке 20.5 показан пример объединения двух куч, в котором встречаются все четыре случая, предусмотренные в тексте процедуры.

Работа процедуры Binomial-Heap-Union начинается с соединения корневых списков куч H_1 и H_2 в единый список H, в котором корневые вершины идут в порядке возрастания их степеней (вызов Binomial-Heap-Merge).

В получившемся списке может быть до двух вершин одинаковой степени. Поэтому мы начинаем соединять деревья равной степени с помощью процедуры Binomial-Link и делаем это до тех пор, пока деревьев одинаковой степени не останется. Этот процесс соответствует сложению столбиком, и время его работы пропорционально числу корневых вершин.

Опишем работу процедуры более подробно. Сначала (строки 1–3) происходит слияние корневых списков биномиальных куч H_1 и H_2 в один корневой список H. Процедура Binomial-Heap-Merge на каждом шаге своей работы сравнивает начала списков H_1 и H_2 и вершину с меньшей степенью добавляет в конец списка H. При этом порядок сохраняется, так что в списке H степени корневых вершин идут в неубывающем порядке. Процедура Binomial-Heap-Merge требует времени $O(m)$, где m — суммарная длина списков H_1 и H_2.

Если кучи H_1 и H_2 пусты, будет возвращена пустая куча (строки 4–5). Далее мы предполагаем, что список H содержит хотя бы одну вершину.

В цикле используются три указателя:

- x указывает на текущую корневую вершину;
- *prev-x* указывает на предшествующую вершину (*sibling*[*prev-x*] $= x$);
- *next-x* указывает на вершину, следующую за x в списке (*sibling*[x] $=$ *next-x*).

Изначально в списке H может быть не более двух вершин данной степени, так как каждая из биномиальных куч H_1 и H_2 содержит не более одной вершины данной степени. Поскольку степени не убывают, вершины одинаковой степени будут соседними.

При сложении двоичных чисел столбиком в одном разряде может быть три единицы (две в слагаемых и одна за счёт переноса). По аналогичным причинам в ходе выполнения процедуры Binomial-Heap-Union в списке H могут оказаться три вершины одинаковой степени.

Поэтому, видя в строке 10 две соседние вершины одинаковой степени, мы проверяем, не идёт ли за ними ещё одна вершина той же степени.

Заметим, что в начале каждой итерации цикла **while** (строки 9–21) ни один из указателей x и *next-x* не равен NIL.

Случай 1, изображенный на рисунке 20.6a, возникает, когда

Рисунок 20.5 Исполнение процедуры Binomial-Heap-Union. (а) Биномиальные кучи H_1 и H_2. (б) Куча H, возникающая после выполнения Binomial-Heap-Merge(H_1, H_2); в переменной x находится первый корень из списка. Обе вершины x и $next\text{-}x$ имеют степень 0, причем $key[x] < key[next\text{-}x]$ (случай 3). (в) После связывания вершин (Binomial-Link) x становится первым из трёх корней одинаковой степени (случай 2) (г) Указатели продвинуты вперёд на одну позицию вдоль корневого списка, x является первым из двух корней одинаковой степени (случай 4). (д) После связывания вершин возникает случай 3. (е) Ещё одно связывание приводит к случаю 1 (степень вершины x равна 3, а степень $next\text{-}x$ равна 4). Продвижение указателей приведет к выходу из цикла, так как $next\text{-}x$ станет равным NIL.

degree[x] $\neq$ *degree*[*next-x*], т. е. когда x является корнем дерева B_k, а *next-x* — корнем дерева B_l для некоторого $l > k$. Данному случаю соответствуют строки 11–12. В этом случае мы просто продвигаем указатели на одну позицию (обновление указателя *next-x* будет выполнено в строке 21).

Случай 2, изображенный на рисунке 20.6б, возникает, когда вершина x является первой среди трёх корневых вершин одинаковой степени (*degree*[x] = *degree*[*next-x*] = *degree*[*sibling*[*next-x*]]). И в этом случае мы продвигаем указатели вдоль списка.

В строке 10 мы проверяем, имеет ли место один из случаев 1 или 2, и в строках 11–12 выполняем продвижение указателей.

В случаях 3 и 4 за вершиной x следует ровно одна вершина той же степени: *degree*[x] = *degree*[*next-x*] $\neq$ *degree*[*sibling*[*next-x*]]. Это может произойти после обработки любого из случаев 1–4, но после случая 2 произойдёт наверняка. Мы связываем вершины x и *next-x*; какая из них остаётся корнем, зависит от того, как соотносятся ключи в этих двух вершинах.

В случае 3 (рис. 20.6в) *key*[x] $\leqslant$ *key*[*next-x*], поэтому вершина *next-x* подвешивается к вершине x. Строка 14 удаляет вершину *next-x* из корневого списка, а строка 15 делает вершину *next-x* ле-

Рисунок 20.6 Четыре случая в процедуре Binomial-Heap-Union. На всех рисунках вершина x — корень дерева B_k и $l > k$. (а) Случай 1: $\mathit{degree}[x] \neq \mathit{degree}[\mathit{next}\text{-}x]$. Указатели продвигаются на одну позицию. (б) Случай 2: $\mathit{degree}[x] = \mathit{degree}[\mathit{next}\text{-}x] = \mathit{degree}[\mathit{sibling}[\mathit{next}\text{-}x]]$. И в этом случае указатели продвигаются, и при следующей итерации цикла возникнет случай 3 или 4. (в) Случай 3: $\mathit{degree}[x] = \mathit{degree}[\mathit{next}\text{-}x] \neq \mathit{degree}[\mathit{sibling}[\mathit{next}\text{-}x]]$, и $\mathit{key}[x] \leqslant \mathit{key}[\mathit{next}\text{-}x]$. Мы удаляем вершину $\mathit{next}\text{-}x$ из корневого списка и подвешиваем её к вершине x, получая дерево B_{k+1}. (г) Случай 4: $\mathit{degree}[x] = \mathit{degree}[\mathit{next}\text{-}x] \neq \mathit{degree}[\mathit{sibling}[\mathit{next}\text{-}x]]$, и $\mathit{key}[\mathit{next}\text{-}x] < \mathit{key}[x]$. Мы удаляем вершину x из корневого списка и подвешиваем её к вершине $\mathit{next}\text{-}x$, опять-таки получая дерево B_{k+1}.

вым ребёнком вершины x.

В случае 4 (рис. 20.6г) меньший ключ имеет вершина *next-x*, поэтому вершина x подвешивается к вершине *next-x*. Строки 16–18 удаляют вершину x из корневого списка; важно различать, является ли вершина x первой в списке (строка 17) или нет (строка 18). Строка 19 делает вершину x левым ребёнком вершины *next-x*, а строка 20 обновляет x для следующей итерации.

В случаях 3 и 4 образовалось новое B_{k+1}-дерево, на которое указывает x. За ним может находиться от нуля до двух деревьев такого же размера. На следующей итерации цикла это приведёт к случаям 1, 3–4 и 2 соответственно.

Время работы процедуры Binomial-Heap-Union есть $O(\lg n)$, где n — суммарное число вершин в кучах H_1 и H_2: в самом деле, размеры корневых списков в кучах H_1 и H_2 суть $O(\lg n)$ (количество элементов в обеих кучах не больше n), и после слияния в списке H будет $O(\lg n)$ элементов. И время работы процедуры Binomial-Merge, и число итераций цикла в строках 9–21 пропорционально начальной длине списка H. Каждая итерация требует времени $O(1)$, так что общее время действительно есть $O(\lg n)$.

Добавление вершины

Следующая процедура добавляет элемент x в биномиальную кучу H. Предполагается, что элемент x уже размещён в памяти, и поле ключа $key[x]$ заполнено.

```
Binomial-Heap-Insert($H, x$)
  $H' \leftarrow$ Make-Binomial-Heap()
  $p[x] \leftarrow$ nil
  $child[x] \leftarrow$ nil
  $sibling[x] \leftarrow$ nil
  $degree[x] \leftarrow 0$
  $head[H'] \leftarrow x$
  $H \leftarrow$ Binomial-Heap-Union($H, H'$)
```

Идея проста: за время $O(1)$ мы создаём биномиальную кучу H' из одной вершины x и за время $O(\lg n)$ объединяем её с биномиальной кучей H, состоящей из n вершин. (После этого память, временно отведённая для H', освобождается). Реализация процедуры Binomial-Heap-Insert без ссылки на Binomial-Heap-Union составляет упр. 20.2-8.

Удаление вершины с минимальным ключом

Следующая процедура изымает из биномиальной кучи H вершину с минимальным ключом и возвращает указатель на изъятую вершину. Поскольку минимальный элемент находится в корневом списке, найти его легко; после его удаления соответствующее дерево рассыпается в набор биномиальных деревьев меньшего размера, который надо объединить с оставшейся частью кучи.

```
Binomial-Heap-Extract-Min($H$)
в корневом списке $H$ найти вершину $x$ с минимальным
        ключом и удалить $x$ из корневого списка
$H' \leftarrow$ Make-Binomial-Heap()
обратить порядок в списке детей вершины $x$
        и поместить в $head[H']$ указатель на начало
        получившегося списка
$H \leftarrow$ Binomial-Heap-Union($H, H'$)
return $x$
```

Работа процедуры показана на рисунке 20.7. Все действия выполняются за время $O(\lg n)$, так что общее время работы процедуры Binomial-Heap-Extract-Min есть $O(\log n)$.

Уменьшение ключа

Следующая процедура уменьшает ключ элемента x биномиальной кучи H, присваивая ему новое значение k (если k больше, чем текущее значение ключа, выдаётся сообщение об ошибке).

```
Binomial-Heap-Decrease-Key($H, x, k$)
 1  if $k > key[x]$
 2     then error "новое значение ключа больше текущего"
 3  $key[x] \leftarrow k$
 4  $y \leftarrow x$
 5  $z \leftarrow p[y]$
 6  while $z \neq$ NIL and $key[y] < key[z]$
 7       do обмен $key[y] \leftrightarrow key[z]$
 8          ▷ (вместе с дополнительной информацией).
 9          $y \leftarrow z$
10          $z \leftarrow p[y]$
```

В данной процедуре используется тот же приём, что и в главе 7: вершина, ключ которой был уменьшен, *"всплывает"* наверх (рис. 20.8).

После проверки корректности входных данных (строки 1–2) в вершине x меняется значение ключа. После строк 4 и 5 перемен-

Рисунок 20.7 Работа процедуры Binomial-Heap-Extract-Min. (а) Исходная биномиальная куча H. (б) Корневая вершина x, имеющая минимальный ключ, удалена из корневого списка H. Она была корнем B_k-дерева, а её дети были корнями B_{k-1}-, B_{k-2}-, …, B_0-деревьев. (в) Обращая порядок, мы собираем потомков вершины x в биномиальную кучу H'. (г) Результат соединения куч H и H'.

Рисунок 20.8 Работа процедуры Binomial-Heap-Decrease-Key. (а) Перед началом цикла (ключ вершины y был уменьшен и стал меньше ключа родительской вершины z). (б) Перед второй итерацией цикла. Произошёл обмен ключами, указатели y и z продвинуты вверх, но свойство порядка пока нарушено. (в) После следующего обмена и сдвига указателей y и z вверх свойство упорядоченности выполнено, и цикл прекращается.

ная y содержит единственную вершину, в которой ключ может быть меньше ключей предков, а z — её родителя. Начинается цикл: если $key[y] \geqslant key[z]$ или если вершина y является корневой, то дерево упорядочено. В противном случае мы меняем местами содержимое вершин y и z, тем самым сдвигая место нарушения вверх по дереву, что отражено в строках 9–10.

Процедура Binomial-Heap-Decrease-Key выполняется за время $O(\lg n)$, поскольку глубина вершины x есть $O(\lg n)$ (утверждение 2 леммы 20.1), а при каждой итерации цикла **while** мы поднимаемся вверх.

Удаление вершины

Удаление вершины сводится к двум предыдущим операциям: мы уменьшаем ключ до $-\infty$ (специальное значение, про которое мы предполагаем, что оно меньше всех ключей), а затем удаляем вершину с минимальным ключом.

В процессе выполнения процедуры Binomial-Heap-Decrease-Key специальное значение $-\infty$ (единственное в куче) всплывает вверх, откуда и удаляется процедурой Binomial-Heap-Extract-Min.

```
Binomial-Heap-Delete($H, x$)
1  Binomial-Heap-Decrease-Key($H, x, -\infty$)
2  Binomial-Heap-Extract-Min($H$)
```

(В упр. 20.2-6 предлагается переписать процедуру Binomial-Heap-Delete без использования дополнительного значения $-\infty$.)

Процедура Binomial-Heap-Delete выполняется за время $O(\lg n)$.

Упражнения

20.2-1 Объясните, почему слияние двоичных куч в смысле главы 7 требует времени $\Theta(n)$.

20.2-2 Напишите процедуру Binomial-Heap-Merge.

20.2-3 Нарисуйте биномиальную кучу, которая получается при добавлении вершины с ключом 24 к куче рис. 20.7г.

20.2-4 Нарисуйте биномиальную кучу, которая получается при удалении ключа 28 из кучи рис. 20.8в.

20.2-5 Почему процедура Binomial-Heap-Minimum не годится, если некоторые из ключей имеют значение ∞? Как сделать её пригодной и для этого случая?

20.2-6 Не пользуясь специальным значением $-\infty$, перепишите процедуру Binomial-Heap-Delete. (Оценка $O(\log n)$ для времени работы должна сохраниться.)

20.2-7 В чём состоит упоминавшаяся связь между соединением биномиальных куч и сложением двоичных чисел? Объясните связь между добавлением вершины в биномиальную кучу и прибавлением единицы к двоичному числу.

20.2-8 Имея в виду соответствие, упомянутое в упр. 20.2-7, перепишите процедуру Binomial-Heap-Insert так, чтобы она добавляла вершину в биномиальную кучу непосредственно, а не вызывала Binomial-Heap-Union.

20.2-9 Покажите, что если располагать вершины в корневых списках в порядке уменьшения (а не увеличения) степеней, то по-прежнему все операции над биномиальными кучами можно реализовать с теми же асимптотическими оценками.

20.2-10 Покажите, что время работы процедур Binomial-Heap-Extract-Min, Binomial-Heap-Decrease-Key и Binomial-Heap-Delete в худшем случае на входах размера n есть $\Omega(\lg n)$.

Объясните, почему время выполнения процедур Binomial-Heap-Insert и Binomial-Heap-Minimum в худшем случае составляет $\overset{\infty}{\Omega}(\lg n)$ (см. зад. 2-5), а не $\Omega(\lg n)$.

Задачи

20-1 2-3-4-кучи

В главе 19 рассматривались 2-3-4-деревья, в которых каждая вершина, не являющаяся корнем или листом, имеет двух, трех или четырёх детей, и все листья располагаются на одинаковой глубине. Определим **2-3-4-кучи** (2-3-4 heaps), над которыми можно производить операции, предусмотренные для сливаемых куч.

Такая куча получается, если в каждом листе записать ключ, а в каждой внутренней вершине хранить наименьшее значение ключей в листьях, являющихся потомками этой вершины. Значения ключей в листьях могут быть любыми (требования упорядоченности нет). В корне хранится высота дерева.

Придумайте способ реализации перечисленных ниже операций над 2-3-4-кучами за время $O(\lg n)$, где n — число элементов 2-3-4-кучи. Операция Union в пункте *е* должна выполняться за время $O(\lg n)$, где n — суммарное число элементов в двух объединяемых кучах.

а. Minimum возвращает указатель на лист с наименьшим ключом.

б. Decrease-Key уменьшает ключ заданного листа x, присваивая ключу заданное значение $k \leqslant key[x]$.

в. Insert добавляет лист x с ключом k.

г. Delete удаляет заданный лист x.

д. Extract-Min изымает лист с наименьшим ключом.

е. Union объединяет две 2-3-4-кучи в одну (исходные кучи пропадают).

20-2 Поиск минимального покрывающего дерева с использованием сливаемых куч

В главе 24 приводятся два алгоритма построения минимального покрывающего дерева для неориентированного графа. Сейчас мы укажем ещё один способ решения этой задачи, использующий сливаемые кучи.

Пусть $G = (V, E)$ — связный неориентированный граф, а $w\colon E \to$

$\mathbb{R}$ — весовая функция, ставящая в соответствие каждому ребру (u, v) его вес $w(u, v)$. Мы хотим найти минимальное покрывающее дерево графа G, т.е. ациклическое множество $T \subseteq E$, которое соединяет все вершины графа, для которого суммарный вес $w(T) = \sum_{(u,v) \in T} w(u, v)$ минимален.

Следующая программа (её корректность доказывается с использованием методов раздела 24.1) строит минимальное покрывающее дерево T. Программа хранит разбиение $\{V_i\}$ множества вершин V, и для каждого из множеств V_i хранит некоторое множество

$$E_i \subseteq \{(u, v) : u \in V_i \text{ или } v \in V_i\}$$

рёбер, инцидентных вершинам из V_i.

```
MST-MERGEABLE-HEAP($G$)
$T \leftarrow \emptyset$
for (для) каждой вершины $v_i \in V[G]$
    do $V_i \leftarrow \{v_i\}$
       $E_i \leftarrow \{(v_i, v) \in E[G]\}$
while (пока) имеется более одного множества $V_i$
    do выбрать любое множество $V_i$
       изъять из $E_i$ ребро $(u, v)$, имеющее минимальный вес
       (будем считать, что $u \in V_i$ и $v \in V_j$)
       if $i \neq j$
         then $T \leftarrow T \cup \{(u, v)\}$
              $V_i \leftarrow V_i \cup V_j$, ($V_j$ пропадает)
              $E_i \leftarrow E_i \cup E_j$
```

Опишите, как реализовать этот алгоритм с помощью операций со сливаемыми кучами, перечисленных в таблице 20.1. Оцените время работы алгоритма, если в качестве сливаемых куч используются биномиальные кучи.

Замечания

Биномиальные кучи были введены в 1978 году Виллемином [196]. Подробно их свойства изучал Браун [36,37].

21 Фибоначчиевы кучи

В главе 20 мы изучали биномиальные кучи, с помощью которых можно реализовать за время $O(\lg n)$ (в худшем случае) операции Insert, Minimum, Extract-Min, Union, а также Decrease-Key и Delete. (Структуры данных, поддерживающие первые четыре из перечисленных операций, называются *"сливаемыми кучами"*.) В этой главе мы рассматриваем фибоначчиевы кучи, которые поддерживают те же шесть операций, но более эффективно: операции, не требующие удаления элементов, имеют учётную стоимость $O(1)$.

Теоретически фибоначчиевы кучи особенно полезны, если число операций Extract-Min и Delete мало́ по сравнению с остальными операциями. Такая ситуация возникает во многих приложениях. Например, алгоритм, обрабатывающий граф, может вызывать процедуру Decrease-Key для каждого ребра графа. Для плотных графов, имеющих много рёбер, переход от $O(\lg n)$ к $O(1)$ в оценке времени работы для операции Decrease-Key может привести к заметному уменьшению общего времени работы. Наиболее быстрые известные алгоритмы для задач построения минимального покрывающего дерева (глава 24) или поиска кратчайших путей из одной вершины (глава 25) существенно используют фибоначчиевы кучи.

К сожалению, скрытые константы в асимптотической записи велики, и использование фибоначчиевых куч редко оказывается целесообразным: обычные двоичные (или k-ичные) кучи на практике эффективнее. С практической точки зрения было бы очень желательно придумать структуру данных с теми же асимптотическими оценками, но с меньшими константами.

Используя биномиальные кучи для хранения набора множеств, мы хранили каждое из множеств набора в нескольких биномиальных деревьях, связанных в список. Сейчас мы будем поступать аналогичным образом, используя фибоначчиевы деревья (которые мы вскоре определим) вместо биномиальных.

Если мы никогда не выполняем операции Decrease-Key и Delete, то возникающие фибоначчиевы деревья будут иметь ту же структуру, что и биномиальные деревья. Но в общем случае фибоначчиевы деревья обладают большей гибкостью, чем биномиальные (из них можно удалять некоторые вершины, откладывая перестройку дерева до удобного случая).

Как и динамические таблицы раздела 18.4, фибоначчиевы кучи являются примером структуры данных, разработанной с учётом амортизационного анализа (см. гл. 18, особенно разд. 18.3 о методе потенциала).

Мы предполагаем, что вы прочли предыдущую главу (о биномиальных кучах). Там была приведена таблица (рис. 20.1), где указаны оценки времени работы различных операций с биномиальными и фибоначчиевыми кучами.

Как и биномиальные кучи, фибоначчиевы кучи не обеспечивают эффективного выполнения поиска (SEARCH). Поэтому передавая вершину в качестве параметра, мы указываем не ключ вершины, а указатель на неё.

В разделе 21.1 мы определим фибоначчиевы кучи, обсудим их представление в программе и введём потенциальную функцию. В разделе 21.2 мы покажем, как реализовать операции, присущие сливаемым кучам, с оценками учётной стоимости, указанными в таблице (рис. 20.1). Оставшиеся две операции (DECREASE-KEY и DELETE) описаны в разделе 21.3. Наконец, в разделе 21.4 мы доказываем лемму, использованную при анализе построенных процедур.

21.1 Строение фибоначчиевой кучи

При использовании фибоначчиевых куч для хранения набора множеств каждое множество занимает несколько деревьев, корни которых связаны в список. Такой конгломерат мы будем называть **фибоначчиевой кучей** (Fibonacci heap). Таким образом, каждая фибоначчиева куча состоит из нескольких деревьев; для каждого хранимого множества отводится своя куча.

В каждом из деревьев, входящих в кучу, выполнено такое свойство: ключ каждой вершины не больше ключей её детей. Деревья, однако, более не обязаны быть биномиальными. Пример фибоначчиевой кучи приведён на рис. 21.1а.

В биномиальных деревьях на детях любой вершины фиксирован порядок; в фибоначчиевых деревьях такого порядка нет (дети любой вершины связаны в круговой двусторонний список, но порядок в нём несуществен). Как показано на рис. 21.1б, каждая вершина x содержит указатель $p[x]$ на своего родителя и указатель $child[x]$ на какого-нибудь из своих детей.

Дети вершины x связаны в двусторонний циклический список, называемый **списком детей** (child list) вершины x. Каждая вершина y этого списка имеет поля $left[y]$ и $right[y]$, указывающие на её соседей

Рисунок 21.1 (а) Фибоначчиева куча, содержащая 5 деревьев и 14 вершин. Пунктирной линией показан корневой список. Минимальная вершина содержит ключ 3. Три отмеченные вершины выделены чёрным цветом. Потенциал этой кучи равен $5 + 2 \cdot 3 = 11$. (б) Та же куча вместе со стрелками, показывающими значения полей p (стрелки вверх), *child* (стрелки вниз) и *left* и *right* (стрелки в стороны). На остальных рисунках этой главы такие стрелки опущены, так как они восстанавливаются однозначно.

в списке (левого и правого). Если вершина y является единственным ребёнком своего родителя, то $left[y] = right[y] = y$.

Двусторонние циклические списки (см. разд. 11.2) удобны по двум причинам. Во-первых, из такого списка можно удалить любую вершину за время $O(1)$. Во-вторых, два таких списка можно соединить в один за время $O(1)$.

Помимо указанной информации, каждая вершина имеет поле $degree[x]$, где хранится её степень (число детей), а также поле $mark[x]$. В этом поле хранится булевское значение. Смысл его таков: $mark[x]$ истинно, если вершина x потеряла ребёнка после того, как она в последний раз сделалась чьим-либо потомком. Мы объясним позже, как и когда это поле используется.

Корни деревьев, составляющих фибоначчиеву кучу, связаны с помощью указателей *left* и *right* в двусторонний циклический список, называемый **корневым списком** (root list).

Доступ к фибоначчиевой куче H осуществляется с помощью атрибута $min[H]$, который указывает на вершину корневого списка с минимальным ключом. Эта вершина называется **минимальной вершиной** (minimum node) кучи. Её ключ будет минимальным ключом в куче, поскольку минимальный ключ фибоначчиева дерева находится в его корне. Порядок вершин в корневом списке значения не имеет. Если фибоначчиева куча H пуста, то $min[H] = \text{NIL}$.

Наконец, атрибут $n[H]$ содержит число вершин в куче H.

Потенциал

При анализе учётной стоимости операций мы используем метод потенциала (раздел 18.3). Пусть $t(H)$ — число деревьев в корневом списке кучи H, а $m(H)$ — количество отмеченных вершин. Потенциал определяется формулой

$$\Phi(H) = t(H) + 2m(H). \tag{21.1}$$

Например, потенциал кучи рис. 21.1 равен $5 + 2 \cdot 3 = 11$. В каждый момент времени в памяти хранится несколько куч; общий потенциал по определению равен сумме потенциалов всех этих куч. В дальнейшем мы выберем *"единицу измерения потенциала"* так, чтобы единичного изменения потенциала хватало для оплаты $O(1)$ операций (формально говоря, мы умножим потенциал на подходящую константу).

В начальном состоянии нет ни одной кучи, и потенциал равен 0. Как и положено (см. разд. 18.3), потенциал всегда неотрицателен.

Максимальная степень

Мы будем предполагать известной некоторую верхнюю границу $D(n)$ для степеней вершин в кучах, которые могут появиться при выполнении наших процедур. (Аргументом функции D является общее число всех вершин в куче, обозначаемое через n.) Если мы используем только операции, предусмотренные для сливаемых куч, то можно положить $D(n) = \lfloor \lg n \rfloor$ (упр. 21.2-3). В разделе 21.3 мы докажем (для общего случая, когда разрешены также операции Decrease-Key и Delete), что $D(n) = O(\lg n)$.

21.2 Операции, предусмотренные для сливаемых куч

Для начала мы будем рассматривать лишь операции Make-Heap, Insert, Minimum, Extract-Min и Union, предусмотренные для сливаемых куч. В этом случае кучи будут представлять собой набор *"неупорядоченных биномиальных деревьев"*, корни которых связаны в циклический список.

Неупорядоченное биномиальное дерево (unordered binomial tree) получается из упорядоченного, если мы перестаём обращать внимание на порядок среди вершин, имеющих общего родителя. Другими словами, неупорядоченное биномиальное дерево U_0 состоит из единственной вершины, а неупорядоченное биномиальное дерево U_k

получается из двух экземпляров деревьев U_{k-1}, если корень одного добавить к числу детей корня другого. (Таким образом, дети корня в дереве U_k являются вершинами деревьев $U_0, U_1, \ldots, U_{k-1}$.) Лемма 20.1 остается верной и для неупорядоченных биномиальных деревьев (надо только исключить в свойстве 4 упоминание о порядке).

В частности, степени всех вершин в фибоначчиевой куче размера n, составленной из неупорядоченных биномиальных деревьев, ограничены величиной $D(n) = \lg n$.

В отличие от биномиальных куч, теперь среди входящих в кучу деревьев может быть несколько деревьев с одной и той же степенью корня. Их *"консолидация"* откладывается до момента выполнения операции EXTRACT-MIN, когда деревья с корнями одинаковой степени объединяются.

Создание новой фибоначчиевой кучи

Процедура MAKE-FIB-HEAP создаёт и возвращает объект H, для которого $n[H] = 0$ и $min[H] =$ NIL: корневой список этой кучи пуст. При этом $t(H) = 0$ и $m(H) = 0$, так что потенциал кучи равен 0, а суммарный потенциал не меняется. Учётная стоимость операции MAKE-FIB-HEAP равна её фактической стоимости $O(1)$.

Добавление вершины

Следующая процедура добавляет вершину x в фибоначчиеву кучу H (предполагаем, что вершина x уже размещена в памяти и поле ключа $key[x]$ заполнено).

```
FIB-HEAP-INSERT($H, x$)
 1  $degree[x] \leftarrow 0$
 2  $p[x] \leftarrow$ NIL
 3  $child[x] \leftarrow$ NIL
 4  $left[x] \leftarrow x$
 5  $right[x] \leftarrow x$
 6  $mark[x] \leftarrow$ FALSE
 7  соединить полученный корневой список
            (состоящий из вершины $x$) с корневым списком кучи $H$
 8  if $min[H] =$ NIL or $key[x] < key[min[H]]$
 9     then $min[H] \leftarrow x$
10  $n[H] \leftarrow n[H] + 1$
```

Строки 1–6 формируют циклический список из единственной вершины x, и в строке 7 эта вершина добавляется (за время $O(1)$) к корневому списку кучи H, в которой появляется новое одноэлемент-

Рисунок 21.2 Добавление вершины. (а) Фибоначчиева куча H. (б) Та же куча H после добавления вершины с ключом 21. (Эту вершину сделали одноэлементным деревом, а затем добавили в корневой список слева от минимальной вершины, которая в данном случае осталась минимальной.)

ное дерево. Вершина x не имеет потомков и не отмечена. В строках 8–9 обновляется (если необходимо) указатель на минимальную вершину. Наконец, строка 10 увеличивает значение $n[H]$.

На рис. 21.2 показано добавление вершины с ключом 21 в фибоначчиеву кучу рис. 21.1.

В отличие от процедуры Binomial-Heap-Insert, процедура Fib-Heap-Insert не пытается соединять деревья с одинаковой степенью вершины. Если выполнить подряд k операций Fib-Heap-Insert, то в корневой список будут добавлены k деревьев по одной вершине в каждом.

Найдём учётную стоимость операции Fib-Heap-Insert. Её фактическая стоимость есть— $O(1)$, и увеличение потенциала также есть $O(1)$ (в корневой список добавилась одна вершина). Таким образом, учётная стоимость составляет $O(1)$.

Поиск минимальной вершины

Указатель на неё хранится в $min[H]$, так что фактическая стоимость этой операции есть $O(1)$. Потенциал при этом не меняется, так что и учётная стоимость есть $O(1)$.

Соединение двух фибоначчиевых куч

Следующая процедура из двух фибоначчиевых куч H_1 и H_2, делает одну (при этом исходные кучи исчезают).

```
Fib-Heap-Union($H_1, H_2$)
1  $H \leftarrow$ Make-Fib-Heap()
2  $min[H] \leftarrow min[H_1]$
3  соединить корневой список $H_2$ с корневым списком $H$
4  if ($min[H_1] =$ NIL) or ($min[H_2] \neq$ NIL and $min[H_2] < min[H_1]$)
5      then $min[H] \leftarrow min[H_2]$
6  $n[H] \leftarrow n[H_1] + n[H_2]$
7  освободить память, занятую под заголовки объектов $H_1$ и $H_2$
8  return $H$
```

Строки 1–3 объединяют корневые списки куч H_1 и H_2 в корневой список новой кучи H. Строки 2, 4 и 5 заполняют $min[H]$, а строка 6 устанавливает $n[H]$ равным суммарному количеству вершин. Объекты H_1 и H_2 освобождаются в строке 7, а строка 8 возвращает результирующую фибоначчиеву кучу H. Отметим, что (как и в процедуре Fib-Heap-Insert) соединения деревьев не происходит. Потенциал не меняется (общее число вершин в корневых списках и общее число помеченных вершин остаётся тем же). Поэтому учётная стоимость операции Fib-Heap-Union равна её фактической стоимости, т.е. $O(1)$.

Изъятие минимальной вершины

Именно при этой операции происходит преобразование структуры кучи (разные деревья соединяются в одно), поэтому она существенно сложнее предыдущих операций этого раздела. План действий таков: после изъятия минимальной вершины то дерево, где она была корнем, рассыпается в набор своих поддеревьев, которые добавляются к корневому списку. Затем запускается процедура Consolidate, соединяющая деревья, после чего в корневом списке остаётся не более одного дерева каждой степени.

Мы считаем, что при удалении вершины из связанного списка (строка 6) поля *left* и *right* этой вершины остаются неизменными (но поля её соседей, которые указывали на эту вершину, обновляются).

```
FIB-HEAP-EXTRACT-MIN($H$)
 1  $z \leftarrow min[H]$
 2  if $z \neq$ NIL
 3     then for (для) каждого ребёнка $x$ вершины $z$
 4              do добавить $x$ в корневой список $H$
 5                 $p[x] \leftarrow$ NIL
 6          удалить $z$ из корневого списка $H$
 7          if $z = right[z]$
 8             then $min[H] \leftarrow$ NIL
 9             else $min[H] \leftarrow right[z]$
10                  CONSOLIDATE($H$)
11          $n[H] \leftarrow n[H] - 1$
12  return $z$
```

Процедура Fib-Heap-Extract-Min перемещает всех детей удаляемой (минимальной) вершины в корневой список кучи H, а саму минимальную вершину удаляет из корневого списка. Затем производится уплотнение корневого списка с помощью процедуры Consolidate.

В строке 1 указатель на минимальную вершину сохраняется в переменной z (этот указатель возвращается процедурой в строке 12). Если $z =$ NIL, исходная куча пуста, и работа заканчивается. В противном случае мы удаляем z из корневого списка H (строка 6), предварительно поместив в него всех детей вершины z (строки 3–5). Если после этого $z = right[z]$, то вершина z не только была единственной вершиной в корневом списке, но и не имела потомков, так что теперь остается лишь сделать кучу пустой (строка 8) и возвратить z. В противном случае мы меняем значение указателя $min[H]$ так, чтобы он указывал на какую-либо вершину корневого списка, отличную от z (в данном случае — на вершину $right[z]$). На рис. 21.3б показано состояние кучи рис. 21.3а после выполнения строки 9.

Остаётся вызвать процедуру Consolidate (о которой мы говорим дальше) для **уплотнения** (consolidating) кучи. Уплотнение происходит за счёт того, что два дерева с вершинами одинаковой степени соединяются в одно (его вершина будет иметь на единицу большую степень). Эта операция (процедура Fib-Heap-Link) производится до тех пор, пока в корневом списке не останется вершин одинаковой степени.

При соединении двух вершин с помощью операции Fib-Heap-Link вершина с большим ключом (назовём её y) становится ребёнком вершины с меньшим ключом (назовём её x). При этом $degree[x]$ увеличивается, а вершина y перестаёт быть отмеченной (если была таковой).

Процедура Consolidate использует вспомогательный массив

$A[0\,..\,D(n[H])]$. Позиция с номером i в этом массиве предназначена для хранения указателя на корень фибоначчиева дерева степени i. Вначале массив пуст (все его ячейки содержат NIL). Постепенно в него переписываются вершины из корневого списка. Если при этом оказывается, что ячейка для вершины нужной степени уже занята, то у нас есть два дерева этой степени, они объединяются, и мы пытаемся записать объединённое дерево в следующую ячейку массива A. Если и она занята, то мы снова производим операцию объединения, получаем дерево ещё на единицу большей степени и т. д.

```
CONSOLIDATE($H$)
 1  for $i \leftarrow 0$ to $D(n[H])$
 2       do $A[i] \leftarrow$ NIL
 3  for (для) каждой вершины $w$ корневого списка кучи $H$
 4       do $x \leftarrow w$
 5          $d \leftarrow degree[x]$
 6          while $A[d] \neq$ NIL
 7              do $y \leftarrow A[d]$
 8                 if $key[x] > key[y]$
 9                   then обмен $x \leftrightarrow y$
10                 FIB-HEAP-LINK($H, y, x$)
11                 $A[d] \leftarrow$ NIL
12                 $d \leftarrow d + 1$
13          $A[d] \leftarrow x$
14  $min[H] \leftarrow$ NIL
15  for $i \leftarrow 0$ to $D(n[H])$
16       do if $A[i] \neq$ NIL
17            then добавить $A[i]$ в корневой список $H$
18                 if $min[H] =$ NIL or $key[A[i]] < key[min[H]]$
19                   then $min[H] \leftarrow A[i]$
```

```
FIB-HEAP-LINK($H, y, x$)
1  удалить $y$ из корневого списка кучи $H$
2  включить $y$ в список детей вершины $x$, увеличив $degree[x]$
3  $mark[y] \leftarrow$ FALSE
```

Опишем работу процедуры CONSOLIDATE более подробно. В строках 1–2 массив A заполняется значениями NIL. В цикле **for** (строки 3–13) мы перебираем все корневые вершины w. Каждую из них мы добавляем к массиву A (соединяя её с имеющимися там вершинами, если нужно). Обработка каждой из корневых вершин w может потребовать нескольких операций FIB-HEAP-LINK и заканчивается созданием дерева, корневая вершина x которого может совпадать, а может и не совпадать с w, но в любом случае это дерево содержит вершину w. После этого элемент массива $A[degree[x]]$ устанавливается таким образом, чтобы он указывал на x. В процессе этих

Рисунок 21.3 Работа процедуры Fib-Heap-Extract-Min. (а) Фибоначчиева куча H. (б) Минимальная вершина z удалена из корневого списка; её потомки включены в корневой список. (в)–(д) Массив A и деревья после каждой из первых трёх итераций цикла **for** (строки 3–13 процедуры Consolidate). Корневой список просматривается слева направо по кругу, начиная с $min[H]$. (е)-(з) Следующая итерация цикла **for**. Рис. (е) показывает состояние после первого исполнения тела цикла **while**: вершина с ключом 23 подвешена к вершине с ключом 7, на которую теперь указывает x. Затем вершина с ключом 17 подвешивается к к вершине с ключом 7 (ж), а потом и вершина с ключом 24 подвешивается к вершине с ключом 7 (з). Поскольку ячейка $A[3]$ была свободной, цикл **for** завершается и $A[3]$ указывает на корень полученного дерева. (и)-(л) Состояния после каждой из следующих четырех итераций цикла **while**. (м) Фибоначчиева куча H после преобразования массива A в корневой список и установки нового значения указателя $min[H]$.

преобразований вершина w остаётся потомком вершины x.

Инвариант цикла **while** (строки 6–12) таков: *"$d = degree[x]$; осталось добавить дерево с корнем в x к множеству, представленному массивом A"*. Если при этом $A[d] = \text{NIL}$, то мы выполняем эту операцию (добавление) в строке 13. Если же $A[d] \neq \text{NIL}$, то мы имеем два дерева степени d с корнями в x и $y = A[d]$, и в строках 8–10 соединяем их в одно дерево с корнем в x (одновременно очищая ячейку $A[d]$ и сохраняя инвариант).

Различные стадии этого процесса показаны на рис. 21.3.

После этого остаётся преобразовать массив A в корневой список: в строке 14 мы создаём пустой список, а в цикле в строках 15–19 добавляем в него по очереди все вершины, имеющиеся в массиве A. Результат показан на рис. 21.3м.

На этом процесс уплотнения заканчивается, и управление возвращается в процедуру FIB-HEAP-EXTRACT-MIN, которая уменьшает $n[H]$ на 1 и возвращает указатель на изъятую вершину z (строки 11–12).

Заметим, что если перед выполнением операции FIB-HEAP-EXTRACT-MIN все деревья в H были неупорядоченными биномиальными деревьями, то и после выполнения операции H будет состоять из неупорядоченных биномиальных деревьев. В самом деле, соединение двух неупорядоченных биномиальных деревьев с одинаковыми степенями корня (процедура Fib-Heap-Link) даёт (неупорядоченное) биномиальное дерево (степень его корня на единицу больше).

Нам осталось показать, что учётная стоимость изъятия минимальной вершины из n-элементной фибоначчиевой кучи есть $O(D(n))$. Посмотрим, какие операции нам пришлось выполнить. Прежде всего нужно $O(D(n))$ действий, чтобы поместить всех потомков удаляемой вершины z в корневой список. Начальный и конечный этапы работы процедуры Consolidate (строки 1–2 и 15–19) также требует времени $O(D(n))$. Остаётся учесть вклад цикла **for**, расположенного в строках 3–13. Это число оценивается сверху как $O(D(n))$ (размер массива A) плюс константа, умноженная на число обращений к процедуре Fib-Heap-Link. Но при каждом таком обращении два дерева сливаются, что приводит в итоге к уменьшению длины корневого списка на 1 и к уменьшению потенциала по крайней мере на 1 (число отмеченных вершин может уменьшиться, но не увеличиться). Так что умножив потенциал на подходящую константу, мы можем считать, что учётная стоимость операции удаления есть $O(D(n))$.

Другими словами, операции связывания деревьев одинаковой степени (которых может быть много, если корневой список длинный) оплачиваются как раз за счёт уменьшения длины корневого списка!

Упражнения

21.2-1 Нарисуйте фибоначчиеву кучу, которая получится в результате изъятия минимальной вершины (с помощью процедуры Fib-Heap-Extract-Min) из кучи рис. 21.3м.

21.2-2 Докажите, что лемма 20.1 остаётся верной для неупорядоченных биномиальных деревьев, если свойство 4 заменить на свойство 4′: корень неупорядоченного биномиального дерева U_k имеет степень k, которая является максимальной степенью вершины в дереве; дети корня являются вершинами деревьев $U_0, U_1, \ldots, U_{k-1}$ (в некотором порядке).

21.2-3 Покажите, что если выполняются лишь операции, описанные в разделе 21.2, то в куче с n вершинами все вершины имеют степень не больше $\lfloor \lg n \rfloor$.

21.2-4 Профессор придумал новую структуру данных, основанную на фибоначчиевых кучах: операции выполняются как описано в разделе 21.2, но только после добавления вершины и соединения двух куч сразу же производится уплотнение корневого списка. Каково время выполнения различных операций над такими кучами в худшем случае? Будет ли такая структура данных чем-то новым?

21.2-5 Будем считать, что ключи можно только сравнивать (их внутренняя структура нам недоступна). Можно ли тогда реализовать операции над сливаемыми кучами так, чтобы каждая из них имела учётную стоимость $O(1)$? (Указание: используйте оценки для времени сортировки.)

21.3 Уменьшение ключа и удаление вершины

В этом разделе мы покажем, как реализовать операцию уменьшения ключа заданной вершины с учётной стоимостью $O(1)$, а также операцию удаления вершины из фибоначчиевой кучи с учётной стоимостью $O(D(n))$ (где n — число вершин в куче), а $D(n)$ — оценка для максимальной степени вершины.

После этих операций входящие в кучу деревья перестают быть биномиальными, но не слишком далеко отклоняются от них, так что максимальная степень вершины остаётся равной $O(\lg n)$. Таким образом, операции FIB-HEAP-EXTRACT-MIN и FIB-HEAP-DELETE имеют учётную стоимость $O(\lg n)$.

Уменьшение ключа

В следующей процедуре уменьшения ключа (FIB-HEAP-DECREASE-KEY) мы предполагаем, что после изъятия вершины из списка ссылка на вершину-ребёнка не меняется (так что процедура CUT вырезает целое поддерево, см. ниже)

```
FIB-HEAP-DECREASE-KEY$(H, x, k)$
if $k > key[x]$
   then error "новое значение ключа больше старого"
$key[x] \leftarrow k$
$y \leftarrow p[x]$
if $y \neq$ NIL and $key[x] < key[y]$
   then CUT$(H, x, y)$
        CASCADING-CUT$(H, y)$
if $key[x] < key[min[H]]$
   then $min[H] \leftarrow x$
```

```
CUT$(H, x, y)$
удалить $x$ из списка детей вершины $y$, уменьшив $degree[y]$ на 1
добавить $x$ в корневой список кучи $H$
$p[x] \leftarrow$ NIL
$mark[x] \leftarrow$ FALSE
```

```
Cascading-Cut($H, y$)
1  $z \leftarrow p[y]$
2  if $z \neq$ NIL
3     then if $mark[y] =$ FALSE
4             then $mark[y] \leftarrow$ TRUE
5             else Cut($H, y, z$)
6                  Cascading-Cut($H, z$)
```

Процедура Fib-Heap-Decrease-Key работает так. Сначала (строки 1–3) проверяется, действительно ли новое значение ключа меньше старого. Если да, то старое значение заменяется на новое. Возможно, новое значение по-прежнему больше значения в вершине-родителе, тогда всё в порядке. Если же нет, то в строках 6–7 поддерево с корнем x вырезается и переносится в корневой список с помощью процедур *"каскадного вырезания"*.

Идея тут проста: мы не хотим позволять вершине всплывать до корня (напомним: нам нужна оценка $O(1)$ для учётной стоимости). Приходится её вырезать целиком и помещать в корневой список. От этого растёт потенциал, но всего на $O(1)$, так что это не страшно. Но мы должны следить за структурой дерева, поскольку хотим иметь логарифмическую оценку на максимальную степень вершины ($D(n) = O(\lg n)$). Забегая вперёд, отметим, что высота дерева не обязана быть логарифмической (см. упр. 21.4-1).

Мы будем следить, чтобы у одной и той же вершины, не входящей в корневой список, не удалялось несколько детей. Для этого используются пометки (поле *mark*): после удаления ребёнка (перенесения его в корневой список с помощью процедуры Cut) вершина делается отмеченной, если она ранее не была отмеченной и не была корнем (строки 3–4 процедуры Cascading-Cut). Если же вершина, у которой удалён ребёнок, уже была отмеченной, то она сама переносится в корневой список, а для её родителя повторяется та же процедура.

После выполнения операций вырезания остаётся только скорректировать атрибут $min[H]$. Заметим, что минимальной может быть либо вершина с уменьшенным ключом, либо прежняя минимальная вершина.

Таким образом, жизненный цикл вершины выглядит так. Сначала она добавляется в дерево, попадая в его корневой список. При выполнении операции Consolidate деревья в корневом списке объединяются. При этом вершина может либо остаться в корневом списке, приобретя нового ребёнка (если её ключ меньше ключа другой вершины, с которой она объединяется), либо стать ребёнком другой вершины корневого списка. Вершина корневого списка может не только приобретать детей, но и терять их (процедура Cut); отметим, что при этом она не становится помеченной (строка 4 процедуры Cascading-Cut выполняется, лишь если выполнено условие в строке 2).

В какой-то момент вершина исключается из корневого списка, становясь ребёнком другой вершины при выполнении процедуры Consolidate. При этом её пометка (если она была) удаляется (строка 3 процедуры Fib-Link). С этого момента новых детей у неё не прибавляется, но может быть удалён один ребёнок, отчего она станет помеченной. При удалении второго ребёнка вершина вновь перемещается в корневой список, становясь непомеченной. Другой способ вернуться в корневой список — оказаться ребёнком изымаемой вершины при операции Fib-Heap-Extract-Min (при этом пометка не меняется).

После возвращения в корневой список к вершине вновь могут добавляться дети (при операции Consolidate). Дети могут и удаляться (операция Cut). В какой-то момент вершина снова может оказаться ребёнком другой вершины корневого списка, и так далее — до тех пор, пока эта вершина не будет изъята из дерева (или не будет уменьшен ключ).

На рис. 21.4 показаны две операции Fib-Heap-Decrease-Key, первая из которых не вызывает цепочки операций Cut, а вторая вызывает.

Покажем, что учётная стоимость операции Fib-Heap-Decrease-Key составляет $O(1)$. Начнем с определения фактической стоимости. Процедуры Fib-Heap-Decrease-Key, Cut и Cascading-Cut не содержат циклов, так что время работы пропорционально длине цепочки рекурсивных вызовов (которую мы обозначим через c).

Как при этом меняется потенциал? Цепочка рекурсивных вызовов соответствует цепочке помеченных вершин в дереве, каждая из которых является ребёнком следующей. Эти вершины перемещаются в корневой список и становятся непомеченными. Таким образом, потенциал увеличивается примерно на c за счёт увеличения числа вершин в корневом списке, но и уменьшается примерно на $2c$ за счёт того, что уменьшается число помеченных вершин. (Правда, может появиться новая помеченная вершина, но эта поправка имеет порядок $O(1)$.) В итоге потенциал уменьшается на $c+O(1)$. (Теперь ясно, почему при определении потенциала число помеченных вершин учитывалось со вдвое большим весом, чем число вершин в корневом списке!)

Умножая потенциал на достаточную константу (выбирая большую *" единицу работы"*), можно считать, что уменьшение потенциала компенсирует фактическую стоимость цепочки рекурсивных вызовов, так что учётная стоимость операции Fib-Heap-Decrease-Key есть $O(1)$.

Рисунок 21.4 Два вызова процедуры FIB-HEAP-DECREASE-KEY. (а) Исходная фибоначчиева куча. (б) Ключ 46 уменьшается до 15, соответствующая вершина становится корневой, а её родитель (с ключом 24) — отмеченным. (в)–(д) Ключ 35 уменьшается до 5; вершина стала корневой (в). Её родитель (с ключом 26) был отмечен, так что его приходится также перенести в корневой список (г); затем то же самое происходит и с ключом 24. На этом каскад вырезаний заканчивается, так как вершина с ключом 7 — корневая. (Если бы она не была корневой, то она стала бы отмеченной, и каскад всё равно закончился бы.) После этого остаётся перенести указатель $min[H]$ на новую минимальную вершину (д).

Удаление вершины

Эта операция сводится к двум рассмотренным ранее (мы считаем, что ключ $-\infty$ меньше всех ключей кучи).

```
FIB-HEAP-DELETE($H, x$)
  FIB-HEAP-DECREASE-KEY($H, x, -\infty$)
  FIB-HEAP-EXTRACT-MIN($H$)
```

Аналогичным образом мы поступали и с биномиальными деревьями. Сначала вершина x делается минимальной, а затем удаляется. Учётная стоимость таких действий равна сумме учётной стоимости операции FIB-HEAP-DECREASE-KEY (которая есть $O(1)$) и операции FIB-HEAP-EXTRACT-MIN (которая есть $O(D(n))$).

Упражнения

21.3-1 Каким образом может появиться помеченная вершина в корневом списке?

21.3-2 Докажите оценку $O(1)$ для учётной стоимости операции Fib-Heap-Decrease-Key, используя метод группировки (раздел 18.1).

21.4 Оценка максимальной степени

Для получения обещанных оценок $O(\lg n)$ для учётной стоимости операций Fib-Heap-Extract-Min и Fib-Heap-Delete осталось показать, что максимальная степень $D(n)$, которую может иметь какая-либо вершина в фибоначчиевой куче с n вершинами, не превосходит $O(\lg n)$.

Как показывает упражнение 21.2-3, если все деревья в фибоначчиевой куче являются неупорядоченными биномиальными деревьями, то $D(n) = \lfloor \lg n \rfloor$. Но операции вырезания, которые происходят во время исполнения процедуры Fib-Heap-Decrease-Key, приводят к тому, что деревья в фибоначчиевой куче более не являются биномиальными. Мы покажем, что тем не менее при выполнении описанных операций остаётся в силе оценка $D(n) = O(\lg_2 n)$. Точнее, мы установим, что $D(n) \leqslant \lfloor \log_\varphi n \rfloor$, где $\varphi = (1+\sqrt{5})/2$.

Для каждой вершины фибоначчиевой кучи через $\mathrm{size}(x)$ обозначим число вершин в поддереве с корнем x, считая саму вершину x. (Вершина x не обязана быть корневой вершиной кучи.) Мы покажем, что величина $\mathrm{size}(x)$ экспоненциально зависит от $degree[x]$. (Напомним, что поле $degree[x]$ поддерживается равным степени вершины x.)

Лемма 21.1. *Пусть x — произвольная вершина фибоначчиевой кучи, и пусть $degree[x] = k$. Тогда степени k детей вершины x не меньше $0, 0, 1, 2, 3, \ldots, k-2$, если их расположить в надлежащем порядке.*

Доказательство. Для вершин, не входящих в корневой список, определим *" модифицированную степень"*, которая на единицу больше реальной степени для помеченных (и совпадает с ней для непомеченных). Смысл этого такой: поскольку при удалении ребёнка у вершины делается пометка, то её модифицированная степень не меняется, а добавление детей возможно только для вершин в корневом списке. Таким образом, модифицированная степень есть степень на момент (последнего) выбытия вершины из корневого списка.

Поскольку модифицированная степень отличается от реальной не более чем на 1, достаточно доказать, что у вершины (реальной) степени k дети имеют модифицированную степень не менее $0, 1, 2, \ldots, k-1$. Ясно, что это свойство сохраняется при удалении одного из детей. Кроме того, оно сохраняется при добавлении к вершине степени k другой вершины степени k (при подчинении одной вершины корневого списка другой первая делается непомеченной, так что её реальная степень равна модифицированной).

□

Теперь оценим число потомков для вершины степени k, используя числа Фибоначчи. Напомним, что k-е число Фибоначчи F_k определяется так:

$$F_k = \begin{cases} 0 & \text{если } k = 0, \\ 1 & \text{если } k = 1, \\ F_{k-1} + F_{k-2} & \text{если } k \geqslant 2. \end{cases}$$

Лемма 21.2.

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i$$

для любого $k \geqslant 0$.

Доказательство. При $k = 0$ это верно: $1 + \sum_{i=0}^{0} F_i = 1 + F_0 = 1 + 0 = 1 = F_2$.

Рассуждая по индукции, предполагаем, что $F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$. Тогда

$$F_{k+2} = F_k + F_{k+1} = F_k + \left(1 + \sum_{i=0}^{k-1} F_i\right) = 1 + \sum_{i=0}^{k} F_i.$$

□

Теперь мы уже можем оценить число вершин в поддереве, если известна степень его корня. Напомним, что $F_{k+2} \geqslant \varphi^k$, где $\varphi = (1 + \sqrt{5})/2 = 1.61803\ldots$ — *"золотое сечение"* (2.14). (Это неравенство доказано в упр. 2.2-8.)

Лемма 21.3. *Пусть* x *— вершина фибоначчиевой кучи, имеющая степень* k *или больше. Тогда* $\mathrm{size}(x) \geqslant F_{k+2} \geqslant \varphi^k$*, где* $\varphi = (1+\sqrt{5})/2$.

Доказательство. По лемме 21.1 вершина x имеет среди своих детей вершины степени не менее $0, 0, 1, 2, 3, \ldots, k-2$. Рассуждая по индукции, мы можем считать, что для детей доказываемое утверждение верно. (При $k = 0$ число F_{k+2} равно 1 и оценка очевидна.) Складывая число вершин в поддеревьях и прибавляя саму вершину x, получаем, что

$$\begin{aligned}\operatorname{size}(x) &\geqslant (F_2 + F_2 + F_3 + F_4 + \ldots + F_k) + 1 = \\ &= (F_0 + F_1 + F_2 + \ldots + F_k) + 1 = F_{k+2}\end{aligned}$$

(мы используем равенство $F_0 + F_1 = F_2$ и лемму 21.2). □

Следствие 21.4. Максимальная степень $D(n)$ какой-либо вершины в фибоначчиевой куче с n вершинами есть $O(\lg n)$.

Доказательство. Пусть x — произвольная вершина такой кучи и пусть $k = degree[x]$. По лемме 21.3 имеем $n \geqslant \operatorname{size}(x) \geqslant \varphi^k$, остаётся взять логарифм по основанию φ.

□

Упражнения

21.4-1 Профессор утверждает, что высота фибоначчиевой кучи из n элементов не превышает $O(\lg n)$. Покажите, что он ошибается и что существует последовательность операций рассмотренных нами типов, которая приводит к куче, состоящей ровно из одного дерева, являющегося линейной цепью из n вершин.

21.4-2 Изменим правила и будем считать, что вершина перемещается в корневой список, когда она потеряла k своих потомков, где k — некоторая константа (до сих пор мы считали, что $k = 2$). При каких значениях k можно утверждать, что $D(n) = O(\lg n)$?

Задачи

21-1 Другой способ удаления элемента

Профессор предложил следующий вариант процедуры Fib-Heap-Delete, утверждая, что новая процедура работает быстрее в том случае, когда удаляемая вершина — не минимальная.

```
New-Delete($H, x$)
if $x = min[H]$
   then Fib-Heap-Extract-Min($H$)
   else $y \leftarrow p[x]$
        if $y \neq$ nil
           then Cut($H, x, y$)
                Cascading-Cut($H, y$)
        добавить детей вершины $x$ в корневой список кучи $H$
        удалить $x$ из корневого списка кучи $H$
```

а. Профессор говорит, что данная процедура работает быстро, поскольку фактическое время исполнения строки 7 есть $O(1)$. Какое обстоятельство он упускает из виду?

б. Оцените сверху время выполнения процедуры New-Delete для случая $x \neq min[H]$ в терминах $degree[x]$ и числа c вызовов процедуры Cascading-Cut.

в. Пусть H' — фибоначчиева куча, получающаяся в результате выполнения процедуры New-Delete(H, x). Предполагая, что вершина x не является корнем, оцените потенциал кучи H' в терминах величин $degree[x]$, c, $t[H]$ (число деревьев в корневом списке) и $m[H]$ (число отмеченных вершин).

г. Получите оценку для учётной стоимости выполнения New-Delete для случая $x \neq min[H]$. Будет ли она лучше прежней?

21-2 Дополнительные операции над фибоначчиевыми кучами

Мы хотим реализовать ещё две операции, не меняя учётной стоимости ранее рассмотренных операций.

а. Придумайте эффективную реализацию операции Fib-Heap-Change-Key(H, x, k) которая присваивает ключу вершины x новое значение k. Оцените учётную стоимость этой операции (при вашей реализации) для случаев $k < key[x]$, $k = key[x]$, $k > key[x]$.

б. Придумайте эффективную реализацию операции Fib-Heap-Prune(H, r) которая удаляет $\min(r, n[H])$ вершин из H (всё равно каких). Оцените учётную стоимость этой операции (для вашей реализации). (Указание: может потребоваться изменение структуры данных и потенциальной функции.)

Замечания

Фибоначчиевы кучи ввели Фредман и Тарьян [75]. В их статье описаны также приложения фибоначчиевых куч к задачам о кратчайших путях из одной вершины, о кратчайших путях для всех пар вершин, о паросочетаниях с весами и о минимальном покрывающем дереве.

Впоследствии Дрисколл, Сарнак, Слеатор и Тарьян разработали структуру данных, называемую *"relaxed heaps"*, как замену для фибоначчиевых куч. Есть две разновидности такой структуры данных. Одна из них даёт те же оценки учётной стоимости, что и фибоначчиевы кучи. Другая позволяет выполнять операцию Decrease-Key за время $O(1)$ в худшем случае (не учётное), а операции Extract-Min и Delete — за время $O(\lg n)$ в худшем случае. Эта структура данных имеет также некоторые преимущества (по сравнению с фибоначчиевыми кучами) при использовании в параллельных алгоритмах.

22 Системы непересекающихся множеств

В некоторых приложениях полезна следующая структура данных: n элементов разбиты в объединение непустых непересекающихся множеств, причем поддерживаются операции " *объединение*" (два данных множества заменяются на их объединение) и "*найти множество*" (по данному элементу выяснить, в каком из множеств он лежит). В этой главе рассказано, как можно реализовать такую структуру данных.

В разделе 22.1 мы даём точное определение интересующей нас структуры данных и приводим простой пример ее применения. В разделе 22.2 обсуждается простейшая реализация с помощью списков. Раздел 22.3 посвящён более эффективной реализации с помощью леса. Для неё время выполнения m операций немного превосходит $O(m)$ — настолько немного, что для всех практических целей можно считать, что время выполнения m операций есть $O(m)$.

Более точно, время выполнения m операций не превосходит $C(m)m$, где $C(m)$ растёт с ростом m, но очень медленно: коэффициент $C(m)$ можно оценить сверху с помощью так называемой "*обратной функции Аккермана*". Определение обратной функции Аккермана даётся в разделе 22.4. Там же доказывается более слабая (и более просто доказываемая) верхняя оценка времени работы.

22.1 Операции с непересекающимися множествами

Система непересекающихся множеств (disjoint-set data structure) есть набор непересекающихся непустых множеств, в каждом из которых зафиксирован один из элементов — *представитель* (representative). При этом должны поддерживаться следующие операции:

Make-Set(x) ("*создать множество*"). Создаёт новое множество, единственным элементом (и тем самым представителем) которого является x. Поскольку множества не должны пересекаться, требуется, чтобы элемент x не лежал ни в одном из уже имеющихся множеств.

Find-Set(x) (”*найти множество*”). Возвращает указатель на представитель (единственного) множества, содержащего элемент x.

Union(x, y) (”*объединение*”). Применима, если элементы x и y содержатся в различных множествах S_x и S_y, и заменяет эти множества на объединение $S_x \cup S_y$; при этом выбирается некоторый представитель для $S_x \cup S_y$. Сами множества S_x и S_y при этом удаляются.

В некоторых приложениях выбор представителя в множестве должен подчиняться каким-то правилам (например, элементы могут быть упорядочены и представителем является наименьший элемент множества). В любом случае существенно, что представитель множества не меняется, пока само множество остается неизменным.

В этой главе мы оценим время работы операций над системами непересекающихся множеств. Параметрами будут число операций Make-Set (то есть общее число элементов во всех множествах), которое мы обозначим через n, а также суммарное число операций Union, Make-Set и Find-Set, которое мы обозначим через m. Прежде всего заметим, что (по определению) $m \geqslant n$, а также что число операций Union не превосходит $n-1$ (после каждой из них количество множеств уменьшается на единицу).

Пример использования систем непересекающихся множеств

Применим системы непересекающихся множеств к задаче о связных компонентах неориентированного графа (см. раздел 5.4; пример графа с четырьмя связными компонентами дан на рис. 22.1а).

Алгоритм Connected-Components (связные компоненты), приведённый ниже, разбивает множество вершин графа на непересекающиеся множества, соответствующие связным компонентам; после этого можно с помощью процедуры Same-Component выяснить, лежат ли две данные вершины в одной компоненте. Если граф задан заранее (”*режим off-line*”), быстрее найти его связные компоненты с помощью поиска в глубину (упражнение 23.3-9); однако если граф строится постепенно и в любой момент надо уметь отвечать на вопрос, каковы его связные компоненты (”*режим on-line*”), то может оказаться выгоднее применить наш алгоритм, а не проводить заново поиск в глубину после добавления каждого из рёбер.

Ниже $E[G]$ и $V[G]$ обозначают множества соответственно рёбер и вершин графа G.

```
Connected-Components(G)
for (для) каждой вершины v\in V[G]
    do Make-Set(v)
for (для) каждого ребра (u,v) \in E[G]
    do if Find-Set(u)\ne Find-Set(v)
          then Union(u,v)
```

Рис. 22.1 (часть (б) я даже б-м нарисовал)

обработанное ребро	непересекающиеся множества									
вначале	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

Рисунок 22.1 22.1 а) Граф, состоящий из четырех связных компонент: $\{a,b,c,d\}$, $\{e,f,g\}$, $\{h,i\}$ и $\{j\}$. б) Последовательные состояния системы непересекающихся множеств в процессе работы алгоритма CONNECTED-COMPONENTS.

```
Same-Component(u,v)
if Find-Set(u)=Find-Set(v)
  then return true
  else return false
```

Алгоритм CONNECTED-COMPONENTS работает так. Сначала каждая вершина рассматривается как одноэлементное подмножество. Далее для каждого ребра графа мы объединяем подмножества, в которые попали концы этого ребра (рис. 22.1б). Когда все рёбра обработаны, множество вершин разбивается на связные компоненты (упр. 22.1-2). Теперь процедура SAME-COMPONENT определяет, лежат ли две данные вершины в одной связной компоненте, дважды вызвав процедуру FIND-SET.

Упражнения

22.1-1

Следуя образцу рис. 22.1, опишите выполнение алгоритма CONNECTED-COMPONENTS для графа G, у которого $V[G] = \{a,b,c,d,e,f,g,h,i,j,k\}$ и $E[G] = \{\,(d,i),(f,k),(g,i),(b,g),(a,h),(i,j),(d,k),(b,j),(d,f),$ Рёбра обрабатываются в том порядке, в котором они выписаны.

22.1-2

Покажите, что алгоритм CONNECTED-COMPONENTS действительно находит связные компоненты графа.

22.1-3

Алгоритм CONNECTED-COMPONENTS применили к графу с v вершинами и e ребрами, состоящему из k связных компонент. Сколько при этом было вызовов процедур FIND-SET и UNION?

Рис. 22.2
Подпись:
а) Представление двух множеств с помощью списков. Представителем множества $\{b, c, e, h\}$ является элемент c, представителем $\{d, f, g\}$ является f. Каждый объект в списке содержит элемент множества, указатель на следующий элемент и указатель на представителя (то есть на начало списка). б) Результат выполнения операции Union(e, g). Представитель объединения множеств есть f.

22.2 Реализация с помощью списков

Самый простой вариант реализации системы непересекающихся множеств хранит каждое множество в виде списка. При этом представителем множества считается первый элемент списка, и каждый элемент списка содержит ссылки на следующий элемент списка и на первый элемент списка (который считается представителем списка). Для каждого списка мы храним указатели на его первый и последний элементы (второй из них нужен при добавлении элементов в конец списка). Порядок элементов в списке может быть любым. На рис. 22.2 (а) изображены два представленных таким образом множества.

При такой реализации операции Make-Set и Find-Set требуют времени $O(1)$: Make-Set создаёт список из одного элемента, а Find-Set возвращает указатель на начало списка.

Простейшая реализация объединения

При естественной реализации операция Union оказывается дорогостоящей. Выполняя Union(x, y), мы добавляем список, содержащий x, к концу списка, содержащего y (рис. 22.2 b). Представителем нового множества при этом будет начало нового списка, то есть представитель множества, содержащего y. При этом нужно ещё установить правильные указатели на начало списка для всех бывших элементов множества, содержащего x, и время на выполнение этой операции линейно зависит от размера указанного множества.

Легко привести пример, в котором время выполнения квадратично зависит от числа операций (рис. 22.3). Пусть даны n элементов $x_1, x_2, \ldots, x_n$. Выполним операции Make-Set(x_i) для всех $i = 1, 2, \ldots, n$, а затем $n-1$ операций Union(x_1, x_2), Union(x_2, x_3), …, Union(x_{n-1}, x_n). Поскольку стоимость операции Union(x_i, x_{i+1}) пропорциональна i, суммарная стоимость выполнения $2n-1$ опера-

Операция	Число затронутых объектов
Make-set(x_1)	1
Make-set(x_2)	1
$\vdots$	$\vdots$
Make-set(x_n)	1
Union(x_1, x_2)	1
Union(x_2, x_3)	2
Union(x_3, x_4)	3
$\vdots$	$\vdots$
Union(x_{n-1}, x_n)	$n - 1$

Рисунок 22.2 22.3 Время выполнения квадратично зависит от числа операций.

ций будет пропорциональна

$$n + \sum_{i=1}^{n-1} i = \Theta(n^2).$$

Весовая эвристика

Простейшая реализация операции Union работает медленно из-за того, что при добавлении длинного списка к короткому приходится присваивать новые значения большому количеству указателей. Дела пойдут лучше, если поступить так: хранить вместе с каждым списком информацию о числе элементов в нём, а при выполнении операции Union добавлять более короткий список в конец более длинного (если длины равны, порядок может быть любым). Такой приём называется *весовой эвристикой* (weighted-union heuristic). Если объединяемые множества содержат примерно поровну элементов, то большого выигрыша не будет, но в целом, как показывает следующая теорема, мы добиваемся экономии.

Теорема 22-1

Предположим, что система непересекающихся множеств реализована с помощью списков, а в операции Union использована весовая эвристика. Тогда стоимость последовательности из m операций Make-Set, Union и Find-Set, среди которых n операций Make-Set, есть $O(m + n \lg n)$ (подразумевается, что первоначально система непересекающихся множеств была пуста).

Доказательство.

Стоимость каждой из операций Make-Set и Find-Set, а также стоимость сравнения размеров, соединения списков и обновления записи о размере множества (все эти действия выполняются при операции Union), есть $O(1)$, так что суммарная стоимость указанных действий есть $O(m)$.

Остаётся оценить стоимость обновления указателей на начало списка. Для этого мы зафиксируем один из элементов (обозначим его x) и проследим, сколько раз у него этот указатель менялся.

Весовая эвристика гарантирует, что представитель меняется, лишь если x входит в меньшее из объединяемых множеств. В этом случае число элементов в множестве, содержащем x, возрастает по крайней мере вдвое. Поскольку окончательный размер множества, содержащего x, не превосходит n, количество таких удвоений не превосходит $\lceil \lg n \rceil$. Общее количество элементов равно n, так что суммарная стоимость обновления указателей на начало списка есть $O(n \lg n)$, а общее число операций есть $O(m) + O(n \lg n)$.

Упражнения

22.2-1

Напишите процедуры Make-Set, Find-Set и Union, используя реализацию с помощью списков и весовую эвристику. Считайте, что каждый элемент x имеет поля $rep[x]$ (указатель на представителя), а также $last[x]$ (указатель на последний элемент списка) и $size[x]$ (число элементов); два последних поля важны только для элемента, являющегося представителем множества.

22.2-2

Изобразите списки, получающиеся в результате работы приведённой ниже программы, и объясните, какой ответ дадут вызовы Find-Set (если используется представление в виде списков и весовая эвристика).

```
for i \gets 1 to 16
    do Make-Set(x_i)
for i \gets 1 to 15 by 2
    do Union(x_i, x_{i+1})
for i \gets 1 to 13 by 4
    do Union(x_i, x_{i+2})
Union(x_1, x_5)
Union(x_{11}, x_{13})
Union(x_1, x_{10})
Find-Set(x_2)
Find-Set(x_9)
```

22.2-3

Покажите, что при реализации непересекающихся множеств с помощью списков с весовой эвристикой учетную стоимость операций Make-Set и Find-Set можно считать равной $O(1)$, а учётную стоимость Union можно считать равной $O(\lg n)$.

22.2-4

Укажите точную асимптотическую оценку стоимости последовательности операций рис. 22.3 (если используются списки и весовая эвристика).

22.3 Лес непересекающихся множеств

Для системы непересекающихся множеств существует более эффективная реализация (по сравнению с рассмотренными). Именно,

Рис. 22.4

Рисунок 22.3 Рис. 22.4. Лес непересекающихся множеств. а) Деревья, представляющие множества рис. 22.2. Представителями множеств являются элементы c и f. б) Результат операции Union(e, g).

представим каждое множество корневым деревом, в котором вершинами являются элементы множества, а корень является представителем. Получается *лес непересекающихся множеств* (disjoint-set forest). Как видно из рисунка (см. рис. 22.4 (а)), каждая вершина указывает на своего родителя, а корень указывает сам на себя. При наивном программировании операций Find-Set и Union такая реализация будет ничуть не лучше списочной; если, однако использовать эвристики *" объединения по рангу"* и *"сжатия путей"*, то получится самая быстрая (из известных в настоящее время) реализация системы непересекающихся множеств.

Наивные реализации операций выглядят так: Make-Set создает дерево с единственной вершиной, Find-Set(x) состоит в том, что мы идем от x по стрелкам (указывающим на родителя), пока не дойдем до корня (путь, который мы при этом проходим, называется *путь поиска*, по-английски find path), а Union состоит в том, что мы заставляем корень одного из деревьев указывать не на самого себя, а на корень другого дерева (рис. 22.4б).

Две эвристики

Пока что больших преимуществ (по сравнению со списочной реализацией) не видно: например, в результате $n-1$ операции Union может получиться дерево, являющееся цепочкой n вершин. Опишем две эвристики, позволяющих добиться почти линейной оценки времени.

Первая эвристика, называемая *объединением по рангу* (union by rank), напоминает весовую эвристику в списочной реализации: мы объединяем деревья не как попало, а так, чтобы корень *" меньшего"* дерева указывал на корень *"большего"*. Выбор *"большего"* определяется не размером дерева, а специальным параметром –*рангом* (rank) его корня. Ранг определён для каждой вершины x дерева и в первом приближении может рассматриваться как грубая оценка логарифма числа вершин в поддереве с корнем в x. Точное определение ранга будет дано в следующем разделе.

Вторая эвристика, применяемая в операции Find-Set, называется *сжатием путей* (path compression). Она заключается в следующем: после того, как путь поиска от вершины к корню пройден, дерево перестраивается: в каждой из вершин пути указатель устанавливается непосредственно на корень (рис. 22.5). При этом ранги остаются прежними.

Рис. 22.5

Рисунок 22.4 Сжатие путей в процессе операции Find-Set. а) Дерево перед выполнением операции Find-Set(a). Треугольниками обозначены поддеревья с корнями в изображенных вершинах. б) После исполнения Find-Set(a).

22.2.1 Программы

Приводимые ниже программы операций Make-Set, Find-Set и Union включают в себя индуктивное определение ранга, но для удобства мы перескажем его словесно. При создании множества с помощью Make-Set единственной вершине дерева присваивается ранг 0. Операция Find-Set (со сжатием путей) не меняет рангов. При выполнении операции Union с деревьями, ранги корней которых различны, проводится новая стрелка от корня меньшего ранга к корню большего ранга, а ранги опять-таки не меняются. Если, наконец, операция Union проводится с деревьями, ранги корней которых равны, то от одного из корней (все равно, какого) проводится стрелка к другому, и при этом ранг корня объединённого дерева увеличивается на единицу (остальные ранги не меняются). Легко видеть, что определённый таким образом ранг вершины является верхней оценкой для высоты поддерева с корнем в этой вершине.

Ниже $p[x]$ обозначает родителя вершины x, а $rank[x]$ — ее ранг; параметрами процедуры Link, вызываемой из процедуры Union, являются корни двух деревьев.

```
Make-Set(x)
p[x] \gets x
rank[x] \gets 0
```

```
Union(x,y)
Link(Find-Set(x),Find-Set(y))
```

```
Link(x,y)
if rank[x] > rank[y]
   then p[y] \gets x
   else p[x] \gets y
        if rank[x]=rank[y]
           then rank[y] \gets rank[y]+1
```

```
Find-Set(x)
if x \ne p[x]
   then p[x] \gets Find-Set(p[x])
return p[x]
```

Рекурсивная процедура Find-Set работает в два прохода: сначала она идет к корню дерева, а затем проходит этот путь в обратном порядке, ”*переводя стрелки*” у встречающихся вершин на корень дерева. Если x — не корень, то Find-Set вызывает саму себя, но уже с параметром — родителем x, после чего делает родителем x корень дерева, найденный этим новым вызовом Find-Set (строка 2). Если же x — корень, то строка 2 не исполняется, и сразу возвращается указатель на x.

Что дают эвристики

Даже будучи применёнными по отдельности, объединение по рангу и сжатие путей дают выигрыш во времени. Если применить объединение по рангу без сжатия путей, то оценка времени работы будет примерно такой же, как при списочной реализации с весовой эвристикой, а именно $O(m \lg n)$ (m — общее количество операций, n –количество операций Make-Set (упр. 22.4-3). Эта оценка неулучшаема (упр. 22.3-3). Если применить сжатие путей без объединения по рангу, то время исполнения последовательности операций, включающей n операций Make-Set и f операций Find-Set, есть (в худшем случае) $\Theta(f \log_{1+f/n} n)$ при $f \geqslant n$ и $\Theta(n + f \lg n)$ при $f < n$ (эти оценки мы доказывать не будем).

Еще бо́льшая экономия получится, если применить обе эвристики совместно. При этом время работы в худшем случае есть $O(m\alpha(m, n))$, где α — чрезвычайно медленно растущая *" обратная функция Аккермана"*, определяемая ниже в разделе 22.4. В любых мыслимых приложениях выполнено неравенство $\alpha(m, n) \leqslant 4$, так что на практике можно считать время работы линейным по m. В разд. 22.4 мы докажем чуть более слабую оценку $O(m \lg^* n)$.

Упражнения

22.3-1

Сделайте упражнение 22.2-2, пользуясь реализацией с помощью леса со сжатием путей и объединением по рангам.

22.3-2

Напишите нерекурсивный вариант процедуры Find-Set.

22.3-3

Приведите пример последовательности m операций Make-Set, Union и Find-Set (в том числе n операций Make-Set, для которой время работы (при использовании объединения по рангу без сжатия путей) будет $\Omega(m \lg n)$.

22.3-4*

Рассмотрим последовательность m операций Make-Set, Link и Find-Set, в которой все операции Find-Set идут после всех операций Link. Докажите, что при использовании объединения по рангам со сжатием путей время работы этой последовательности операций есть $O(m)$. Что будет, если пользоваться только сжатием путей?

22.4* Объединение по рангам со сжатием путей: анализ

В этом разделе мы дадим определение *"обратной функции Аккермана"* $\alpha(m, n)$, входящей в оценку $O(m\alpha(m, n))$ для стоимости m операций с системой непересекающихся множеств, реализованной с помощью леса с объединением по рангам и сжатием путей

Рисунок 22.5 22.6. Значения $A(i,j)$ для малых i и j.

Переводы: row — строка, column — столбец

Рисунок 22.6 Рис. 22.7. Рост функции Аккермана. В таблице значений (рис. 22.6) соединены равные числа. Масштаб по горизонтали соблюсти невозможно (из-за очень быстрого роста).

(здесь n — число операций Make-Set). Эту оценку мы доказывать не будем, ограничившись более слабой оценкой $O(m \lg^* n)$.

Функция Аккермана и обратная к ней

Функцией Аккермана (Ackermann's function) называется функция $A(i,j)$ двух целых положительных переменных, определённая так:

$$\begin{aligned} A(1,j) &= 2^j, && \text{если } j \geqslant 1;\\ A(i,1) &= A(i-1,2) && \text{если } i \geqslant 2;\\ A(i,j) &= A(i-1,A(i,j-1)) && \text{если } i,j \geqslant 2. \end{aligned}$$

Значения $A(i,j)$ для малых i и j изображены на рис. 22.6.

В таблице значений функции Аккермана каждая следующая строка есть подпоследовательность предыдущей (начинающаяся со второго члена); отсюда следует, что $A(i,j)$ увеличивается при увеличе- нии i или j. Эта функция растёт очень быстро: уже первая строка в таблице её значений растёт экспоненциально, а каждая последующая строка есть (весьма редкая) подпоследовательность предыдущей. Именно, в $k+1$-ой строке таблицы записана последо- вательность, каждый следующий член которой получается из преды- дущего применением функции, записанной в k-ой строке. В частности, во второй строке каждый следующий член есть двойка, возведённая в степень предыдущего члена.

Взаимоотношения между различными строками таблицы значений функции Аккермана схематически изображены на рис. 22.7.

Определим, наконец, обратную функцию Аккермана $\alpha(m,n)$. Строго говоря, α не является обратной к функции Аккермана A (тем более что последняя имеет два аргумента), но можно сказать, что функция α растет столь же медленно, сколь быстро растет A. (Детали определения α существенны для доказательства оценки $O(m\alpha(m,n))$, которое выходит за рамки нашей книги.) Итак, при $m \geqslant n \geqslant 1$ полагаем:

$$\alpha(m,n) = \min\{\, i \geqslant 1 : A(i, \lfloor m/n \rfloor) > \lg n \,\}.$$

Легко видеть, что при фиксированном n функция $\alpha(m,n)$ монотонно убывает с ростом m. Это согласуется с (не доказываемым нами) утверждением о том, что стоимость m операций с непересекающимися множествами, включающих n операций Make-Set, есть $\Theta(m\alpha(m,n))$. В самом деле, эта оценка говорит, что стоимость в расчёте на одну операцию есть $\alpha(m,n)$. Если n фиксировано, а m растёт, то из-за сжатия путей деревья с каждым шагом всё более *" упрощаются"* (всё больше становится вершин, напрямую связанных с корнем), так что стоимость операции падает.

Во всех практических приложениях можно считать, что $\alpha(m,n) \leqslant 4$. В самом деле, поскольку функция Аккермана является возрастающей по каждому переменному и $m \geqslant n$, имеем

$$A(4,\lfloor m/n\rfloor) \geqslant A(4,1) = A(3,2) = \left.2^{2^{\cdot^{\cdot^{2}}}}\right\}17,$$

что много больше числа атомов в наблюдаемой части Вселенной ($\approx 10^{80}$). Поэтому для всех встречающихся на практике значений n имеем $A(4,\lfloor m/n\rfloor) \geqslant \lg n$ и $\alpha(m,n) \leqslant 4$.

Мы не доказываем оценки, связанной с функцией Аккермана, а ограничиваемся более слабой оценкой $O(m \lg^* n)$. Вспомним определение функции $\lg^*$ (с. ??). По определению, $\lg^* 1 = 0$ и

$$\lg^* n = \min \left\{ i : \underbrace{\lg\lg\ldots\lg}_{i \text{ раз}} n \leqslant 1 \right\}$$

для $n > 1$. Неравенство в определении функции $\lg^*$ можно переписать так: $n \leqslant \left.2^{2^{\cdot^{\cdot^{2}}}}\right\}j$ (j — число двоек в формуле).

Оценка $O(m \lg^* n)$, которую мы докажем ниже, слабее, чем сформулированная без доказательства оценка $O(m\alpha(m,n))$, но с практической точки зрения разница невелика. В самом деле, $\lg^* 2^{2^{16}} = \lg^* 2^{65536} = 5$, так что во всех практических вопросах можно считать, что $\lg^* n \leqslant 5$ — вряд ли нам встретится множество, в котором более 2^{65536} элементов.

Свойства рангов

Чтобы доказать оценку $O(m \lg^* n)$ для времени работы с системой непересекающихся множеств, нам понадобятся некоторые свойства рангов. Напомним, что мы работаем с системой непересекающихся множеств, реализованной с помощью леса, с объединением по рангам и сжатием путей (см. разд. 22.3); через m обозначено суммарное число операций Make-Set, Find-Set и Union, а через n — число операций Make-Set.

Лемма 22.2

Для всякой вершины x, кроме корня, $rank[x] < rank[p[x]]$ (напомним, что для корня $p[x] = x$). При создании множества $\{x\}$ имеем $rank[x] = 0$, далее $rank[x]$ может только возрастать и перестаёт меняться, когда x перестаёт быть корнем в своём дереве.

Доказательство. Всё это непосредственно следует из описания процедур Make-Set, Union и Find-Set в разделе 22.3. Подробности мы оставляем читателю (упр. 22.4-1).

Пусть x — вершина одного из наших деревьев. Число вершин в поддереве с корнем x (включая саму x) назовем ее *размером* (size) и обозначим $\text{size}(x)$.

Лемма 22.3

Если x — корень одного из деревьев, то $\text{size}(x) \geqslant 2^{rank[x]}$.

Доказательство.

Достаточно проверить, что каждая из операций Make-Set, Find-Set и Link сохраняет истинность утверждения " $size(x) \geqslant 2^{rank[x]}$ *для всех корней* x". Операция Find-Set не меняет ни размеров корней, ни рангов каких бы то ни было вершин. Операция Make-Set(x) также не нарушает утверждения леммы: не меняя рангов и размеров уже существующих деревьев, она создаёт новое дерево с единственной вершиной, для которой неравенство очевидным образом выполнено.

Рассмотрим операцию Link(x, y) (не ограничивая общности, можно считать, что $rank[x] \leqslant rank[y]$). В результате этой операции ранг или размер может измениться только у вершины y. Если $rank[x] < rank[y]$, то размер y увеличивается, а ранг не меняется, так что неравенство $size(y) \geqslant 2^{rank[y]}$, остаётся верным. Если же $rank[x] = rank[y]$, то, обозначая через $size'[y]$ и $rank'[y]$ размер и ранг y после операции Link, имеем $size'[y] = size[x] + size[y]$ и $rank'[y] = rank[y] + 1$. Отсюда

$$size'[y] = size[x] + size[y] \geqslant 2^{rank[x]} + 2^{rank[y]} = 2^{rank[y]+1} = 2^{rank'[y]}.$$

Лемма 22.4

Число вершин ранга r не превосходит $n/2^r$ (для любого целого $r \geqslant 0$).

Доказательство.

Зафиксируем число r. Всякий раз, когда какой-либо вершине y присваивается ранг r (в строке 2 процедуры Make-Set или в строке 5 процедуры Link), будем привешивать метку ко всем вершинам поддерева с вершиной y. Поскольку вершина y в этот момент является корнем одного из деревьев, лемма 22.3 показывает, что число помечаемых вершин не меньше 2^r. Наши процедуры устроены так, что каждая вершина будет помечена не более одного раза. Стало быть,

$$n \geqslant (\text{число меток}) \geqslant 2^r \cdot (\text{число вершин ранга } r),$$

откуда всё и следует.

Следствие 22.5

Ранг любой вершины не превосходит $\lfloor \lg n \rfloor$.

Доказательство.

См. лемму 22.3 или 22.4

Доказательство оценки для времени работы

При доказательстве оценки нам будет удобнее работать с операцией Link, а не Union. Пусть дана последовательность S_1, состоящая из m_1 операций Make-Set, Find-Set и Union. Заменим каждую операцию Union на две операции Find-Set и одну операцию Link; получится последовательность S_2 из m_2 операций Make-Set, Find-Set и Link.

Лемма 22.6

Если стоимость выполнения последовательности S_2 в худшем случае есть $O(m_2 \lg^* n)$, то стоимость выполнения последовательности S_1 в худшем случае есть $O(m_1 \lg^* n)$, где n — число операций Make-Set.

Доказательство.

Немедленно следует из очевидного неравенства $m_1 \leqslant m_2 \leqslant 3m_1$.

Теорема 22.7

Предположим, что над системой непересекающихся множеств (первоначально пустой) произвели m операций Make-Set, Find-Set и Link, из которых n операций были операциями Make-Set. Тогда при использовании реализации с помощью леса со сжатием путей и объединением по рангам общая стоимость этой последовательности операций есть $O(m \lg^* n)$.

Доказательство.

Стоимость каждой из операций Make-Set и Link есть, очевидно, $O(1)$, а их суммарная стоимость есть тем самым $O(m)$. Займемся операциями Find-Set. Очевидно, стоимость операции Find-Set(x) пропорциональна длине пути поиска от x к корню (до сжатия). Стало быть, нам надо оценить суммарную длину путей поиска, возникавших в процессе выполнения всех операций Find-Set и доказать, что она есть $O(m \lg^* n)$.

В процессе выполнения каждой из операций Find-Set мы двигаемся вверх по одному из деревьев, заканчивая поиск в его корне. Заметим, что на каждом шаге ранг вершины строго возрастает: если u — вершина, встретившаяся на пути поиска, а v — следующая вершина, то есть $v = p[u]$, то $rank[v] > rank[u]$. Вершина u может неоднократно встречаться в путях поиска при выполнении рассматриваемой в теореме последовательности операций. При этом за ней могут (и скорее всего будут) идти разные вершины: если в некоторый момент за u следовала вершина v, которая не была корневой, то после сжатия путей за u будет следовать уже не v, а корень дерева (на момент первого поиска), то есть вершина большего ранга, чем v. (Это простое наблюдение играет к дальнейшем ключевую роль.)

Наблюдая за ростом ранга при переходе от вершины к её родителю, мы отдельно оценить количество шагов, при которых ранг сильно растёт, и количество шагов, когда он растёт не сильно. Выберем в качестве границы некоторую функцию $\beta(k)$, определённую для неотрицательных целых k. Мы предполагаем, что функция β

является монотонно возрастающей и что $\beta(k) > k$ при всех k. Будем говорить, что при переходе от вершины u к её родителю $v = p[u]$ ранг сильно растёт, если $rank[v] \geqslant \beta(rank[u])$. При этом мы исключаем из рассмотрения случай, когда v является корнем — такой случай встречается по разу при выполнении каждой операции Find-Set, и потому общее число таких ситуаций есть $O(m)$.

Начнём с оценки числа шагов, при которых ранг не сильно растёт, и сгруппируем их по вершинам, из которых этот шаг делается. Для вершины ранга k ранги её предка могут меняться от $k+1$ до $\beta(k)$ (после этого ранг растёт сильно). Значит, число таких шагов (из данной вершины ранга k) заведомо не больше $\beta(k)$, поскольку, как мы говорили, каждый новый шаг ведёт в вершину большего ранга, чем предыдущий. Поскольку вершин ранга k не более $n/2^k$, то общее число шагов, при которых ранг не сильно растёт, не превосходит

$$\sum_k \frac{n\beta(k)}{2^k}$$

Если выбрать функцию β так, чтобы ряд $\sum \beta(k)/2^k$ сходился, то общее число шагов такого рода есть $O(n)$.

Прежде чем выбрать функцию β, объясним, как оценить число шагов, при которых ранг сильно растёт. Такие шаги мы сгруппируем не по вершинам, а по путям: на каждом пути поиска таких шагов мало, так как ранг не может многократно сильно расти (он меняется всего лишь от 0 до $\lg n$). Таким образом, на каждом пути число шагов, при котором ранг сильно растёт, не превосходит числа итераций функции β, которые нужно сделать, чтобы дойти от 0 до $\lg n$.

Хотелось бы положить $\beta(k) = 2^k$: тогда число итераций будет примерно равно $\lg^* n$. К сожалению, тогда написанный выше ряд расходится. Придётся взять немного меньшую функцию. Например, положим $\beta(k) = \lceil 1{,}9^k \rceil$, тогда ряд $\sum \lceil 1{,}9^k \rceil / 2^k \leqslant \sum (1{,}9^k + 1)/2^k$ сходится. С другой стороны, число итераций функции β, которые нужно сделать, чтобы от 0 дойти до какого-то числа, возрастёт (по сравнению с функцией $k \mapsto 2^k$) не более чем вдвое, поскольку $\beta(\beta(k)) \geqslant 2^k$, и потому есть $O(\lg^* n)$. Итак, число шагов такого рода для всех m операций есть $O(m \lg *n)$.

Складывая вместе действия всех видов (операции Make-Set и Link, последние шаги в каждом пути, шаги, на которых ранг не сильно растёт, и шаги, на которых ранг сильно растёт), получаем общую оценку $O(n) + O(m \lg^* n) = O(m \lg^* n)$ (напомним, что $m \geqslant n$, так как m — общее число операций, а n — число операций textscMake-Set), что и требовалось доказать.

[Замечание. Доказательство этой теоремы слегка изменено по сравнению с английским оригиналом: в последнем используется функция $\beta(k)$, равная наименьшему члену последовательности

$1, 2, 4, 16, 65536, \ldots$, большему k (каждый член последовательности есть двойка в степени предыдущего члена). Тогда на каждом пути будет не более $\lg^* n$ шагов, на которых ранг сильно растёт, а ряд $\sum \beta(k)/2^k$ сходится, но имеет достаточно медленно растущие частичные суммы.]

Из доказанной теоремы и леммы 22.6 немедленно вытекает

Следствие 22.8

Предположим, что над системой непересекающихся множеств (изначально пустой) произвели m операций Make-Set, Find-Set и Union, n из которых — Make-Set. Тогда при использовании реализации с помощью леса со сжатием путей и объединением по рангам стоимость этой последовательности операций есть $O(m \lg^* n)$.

Упражнения

22.4-1

Докажите лемму 22.2.

22.4-2

Сколько битов нужно, чтобы хранить $size[x]$ для узла x? Тот же вопрос для $rank[x]$.

22.4-3

Пользуясь леммой 22.2 и следствием 22.5, докажите, что стоимость m операций с лесом непересекающихся множеств с использованием объединения по рангам, но без сжатия путей, есть $O(m \lg n)$, где n, как обычно, обозначает количество операций Make-Set.

22.4-4*

Объясните, почему последние шаги путей требовали в нашем рассуждении особого рассмотрения (а на классифицировались в зависимости от роста ранга, как все остальные): приведите пример ситуации, в которой некоторая вершина x входит $\Omega(m)$ раз в путь поиска для операции Find-Set, причём при выходе из неё ранг растёт не сильно. (Как видно из доказательства, это возможно лишь если она по большей части является предпоследней вершиной пути.)

Задачи

22-1 Поиск минимума в режиме off-line.

Задача о поиске минимума в режиме off-line (off-line minimum problem) состоит в следующем. Пусть имеется динамическое множество T, поддерживающее операции Insert(x) (добавить элемент x) и Extract-Min (удалить минимальный элемент). Первоначально множество T пусто, затем выполняется некоторая последовательность из n операций Insert и m операций Extract-Min, причем операции Insert добавляют в множество по одному разу все натуральные числа от 1 до n (не обязательно в порядке возрастания). Требуется по данной последовательности операций Insert(x) и Extract-Min создать массив $extracted[1..m]$, в котором $extracted[i]$ — число, возвращаемое i-ой по счёту операцией Extract-Min.

Говоря о *"режиме off-line"*, имеют в виду, что от нас требуется дать ответ после того, как мы знаем всю последовательность команд (в противоположность этому, в режиме *"on-line"* от нас требовалось бы давать ответ немедленно по поступлении очередной команды, не дожидаясь следующей).

(а) Пусть команды поступали в такой последовательности (мы пишем E вместо Extract-Min и i вместо Insert(i)):

$$4, 8, \mathrm{E}, 3, \mathrm{E}, 9, 2, 6, \mathrm{E}, \mathrm{E}, \mathrm{E}, 1, 7, \mathrm{E}, 5.$$

Как выглядит массив *extracted*?

Алгоритм для решения задачи о минимуме в режиме off-line может выглядеть следующим образом. Пусть S — данная последовательность команд. Представим ее в виде

$$\mathrm{I}_1, \mathrm{E}, \mathrm{I}_2, \mathrm{E}, \mathrm{I}_3, \ldots, \mathrm{I}_m, \mathrm{E}, \mathrm{I}_{m+1},$$

где каждое E обозначает операцию Extract-Min, а каждое I_j обозначает последовательность операций Insert (возможно, пустую). Для каждой последовательности I_j создадим множество K_j, состоящее из чисел, добавляемых в T при операциях Insert из последовательности I_j. А теперь сделаем вот что:

```
Off-Line-Minimum(m,n)
for i \gets 1 to n
    do найти такое $j$, что $i\in K_j$
       if j \ne m+1
          then extracted[j] \gets i
               l \gets (наименьшее число, большее $j$,
                       для которого существует множе-
ство $K_l$)
               K_l\gets K_l\cup K_j (множество $K_j$ исчезает)
return extracted
```

(б) Покажите, что алгоритм Off-Line-Minimum правильно заполняет массив *extracted*.

(в) Реализуйте алгоритм Off-Line-Minimum с помощью системы непересекающихся множеств. Дайте достаточно точную оценку времени работы в худшем случае.

22-2 Определение глубины

В *задаче определения глубины* (depth-determination problem) требуется реализовать лес $\mathcal{F}$, состоящий из корневых деревьев T_i, поддерживающий следующие три операции:

Make-Tree(v) Создает новое дерево с единственной вершиной v.

Find-Depth(v) Возвращает глубину (расстояние до корня) вершины v.

Graft(r, v) (*”прививка”*). Вершина r должна быть корнем одного из деревьев, вершина v должна принадлежать к какому-то другому дереву; дерево с корнем r *”прививается”* к дереву, содержащему v, при этом v становится родителем r.

(а) Пусть мы реализовали эту структуру данных следующим образом. Деревья представляются как в лесе непересекающихся множеств ($p[v]$ — родитель вершины v, если v — не корень, и $p[v] = v$, если v — корень), процедура Graft(r, v) состоит в присваивании $p[r] \leftarrow v$, процедура Make-Tree написана очевидным образом, и, наконец, для нахождения глубины (Find-Depth(v)) мы идем из v в корень и подсчитываем длину пути. Покажите, что при этом стоимость m операций Make-Tree, Graft и Find-Depth в худшем случае есть $\Theta(m^2)$.

Алгоритм можно ускорить, если воспользоваться объединением по рангам и сжатием путей. Заметим, что структура дерева, нужного для сжатия путей, не обязана соответствовать структуре исходного дерева — важно лишь, чтобы можно было восстанавливать информацию о глубине (ребро нового дерева должно хранить информацию о разнице глубин концов ребра в старом дереве).

(б) Реализуйте операцию Make-Tree.

(в) Реализуйте операцию Find-Depth. Ваш алгоритм должен использовать сжатие путей, а его время работы должно быть пропорционально длине пути поиска.

(г) Реализуйте операцию Graft (действуйте по аналогии с алгоритмами Union и Link; корень в строимом дереве не обязан быть корнем в старом смысле).

(д) Дайте точную оценку на стоимость последовательности m операций Make-Tree, Graft и Find-Depth, n из которых — операции Make-Tree (для худшего случая).

22-3 Алгоритм Тарьяна для нахождения наименьшего общего предка в режиме off-line.

Наименьший общий предок (least common ancestor, сокращённо LCA) вершин u и v корневого дерева T есть, по определению, вершина наибольшей глубины среди вершин, являющихся предками как u, так и v. *Задача о нахождении наименьших общих предка в режиме off-line* (off-line least-common-ancestors problem) состоит в следующем. Дано корневое дерево T и некоторое множество P неупорядоченных пар его вершин. Требуется для каждой пары вершин $(u, v) \in P$ найти их наименьшего общего предка.

Ниже приведён алгоритм LCA, решающий эту задачу (наименьшие общие предки всех пар $(u, v) \in P$ будут напечатаны в результате вызова LCA$(root[T])$; вначале все вершины дерева — белые).

```
LCA(u)
  Make-Set(u)
```

```
2  ancestor[Find-Set(u)] \gets u
3  for (для) каждого $v$, являющегося ребёнком $u$
4      do LCA(v)
5         Union(u,v)
6         ancestor[Find-Set(u)] \gets u
7  покрасить u в чёрный цвет
8  for (для) каждой вершины v такой, что $(u,v)\in P$
9      do if вершина v чёрная
10        then print (``Наименьший общий предок '' $u$ `` и '' $v$
                    `` есть '' ancestor[Find-Set(v)]
```

(а) Покажите, что строка 10 исполняется в точности один раз для каждой пары $(u, v) \in P$.

(б) Покажите, что в результате вызова LCA($root[T]$) каждый из последующих вызовов LCA(u) происходит в тот момент, когда количество непересекающихся множеств равно глубине вершины u в дереве T.

(в) Покажите, что в результате вызова LCA($[T]$) будут напечатаны наименьшие общие предки для всех пар $(u, v) \in P$.

(г) Оцените время работы алгоритма LCA в предположении, что система непересекающихся множеств реализована с помощью леса с объединением по рангам и сжатием путей.

Замечания

Многие важные результаты о системах непересекающихся множеств в той или иной мере принадлежат Тарьяну. В частности, именно он установил оценку $O(m\alpha(m, n))$ [186, 188]. Более слабая оценка $O(m \lg^* n)$ была ранее получена Хопкрофтом и Ульманом [4, 103]. В работе [190] Тарьян и ван Леувен обсуждают различные варианты сжатия путей, в том числе алгоритмы, работающие *"за один проход"* (иногда они работают быстрее *"двухпроходных"*). Габов и Тарьян [76] показали, что в некоторых приложениях операции с непересекающимися множествами можно выполнить за время $O(m)$.

В работе [187] Тарьян показал, что при некоторых дополнительных условиях стоимость операций с непересекающимися множествами не может быть ниже, чем $O(m\alpha(m, n))$, какую реализацию мы бы ни избрали. Фредман и Сакс [74] показали, что, кроме того, в худшем случае эти операции требуют обращения к $\Omega(m\alpha(m, n))$ словам длиною в $\lg n$ битов.

23 Алгоритмы на графах

Введение

Графы встречаются в сотнях разных задач, и алгоритмы обработки графов очень важны. В этой части книги мы рассмотрим несколько основных алгоритмов обработки графов.

В главе 23 рассматриваются способы представления графа в программе, а также различные варианты обхода графа (поиск в ширину и в глубину). Приводятся два применения поиска в глубину: топологическая сортировка ориентированного графа без циклов и разложение ориентированного графа в сумму сильно связных компонент.

В главе 24 рассматривается задача о покрывающем дереве (наборе рёбер, связывающем все вершины графа) минимального веса. (Мы предполагаем, что каждое ребро графа имеет некоторый положительный вес.) Для этой задачи применимы жадные алгоритмы (см. главу 18).

В главах 25 и 26 рассматривается задача о кратчайших путях. Вновь каждому ребру приписано некоторое число, но теперь оно называется длиной ребра. Требуется найти кратчайшие пути из одной вершины во все остальные (глава 25) или кратчайшие пути из каждой вершины в каждую (глава 26).

Наконец, в главе 27 расматривается задаче о максимальном потоке вещества через сеть труб ограниченной пропускной способности. Каждое ребро рассматривается как труба некоторой пропускной способости. В некоторой вершине находится источник вещества, а в другой — потребитель. Спрашивается, какой поток вещества можно передавать от источника к потребителю, не превышая пропускной способности труб. Эта задача важна, поскольку к ней сводится многие интересные задачи.

Мы будем оценивать время обработки заданного графа $G = (V, E)$ в зависимости от числа его вершин ($|V|$) и рёбер ($|E|$); мы будем для краткости писать V и E вместо $|V|$ и $|E|$. В программах множество вершин граф G мы будем обозначать $V[G]$, а множество рёбер — $E[G]$.

23.1 Основные алгоритмы на графах

В этой главе описаны основные способы представления графов и алгоритмы обхода графов. Обходя граф, мы двигаемся по рёбрам и проходим все вершины. При этом накапливается довольно много информации, которая полезна для дальнейшей обработки графа, так что обход графа входит как составная часть во многие алгоритмы.

В разделе 23.1 обсуждаются два основных способа представления графа в памяти компьютера — с помощью списков смежных вершин и с помощью матрицы смежности. Раздел 23.2 описывает алгоритм поиска в ширину и соответствующее этому алгоритму дерево. В разделе 23.3 рассматривается другой порядок обхода вершин и доказываются свойства соответствующего алгоритма (называемого поиском в глубину). В разделе 23.4 мы используем разработанные методы для решения задачи о топологической сортировке ориентированного графа без циклов. Другое применение поиска в глубину (отыскания сильно связных компонент ориентированного графа) описано в разделе 23.5.

23.1.1 Представление графов

Есть два стандартных способа представить граф $G = (V, E)$ — как набор списков смежных вершин или как матрицу смежности. Первый обычно предпочтительнее, ибо даёт более компактное представление для *разреженных* (sparse) графов — тех, у которых $|E|$ много меньше $|V|^2$. Большинство издагаемых нами алгоритмов используют именно это представление. Однако в некоторых ситуациях удобнее пользоваться матрицей смежности — например, для *плотных* (dense) графов, у которых $|E|$ сравнимо с $|V|^2$. Матрица смежности позволяет быстро определить, соединены ли две данные вершины ребром. Два алгоритма отыскания кратчайших путей для всех пар вершин, описанные в главе 26, используют представление графа с помощью матрицы смежности.

Представление графа $G = (V, E)$ в виде *списков смежных вершин* (adjancency-list representation) использует массив Adj из $|V|$ списков — по одному на вершину. Для каждой вершины $u \in V$ список смежных вершин $Adj[u]$ содержит в произвольном порядке (указатели на) все смежные с ней вершины (все вершины v, для которых $(u, v) \in E$).

На рис. 23.1 (b) показано представление неориентированного графа рис. 23.1 (a) с помощью списков смежных вершин. Аналогичное представление для ориентированного графа рис. 23.2 (a) изображено на рис. 23.2 (b).

Рисунок 23.1 23.1 Два представления неориентированного графа. (a) Неориентированный граф G с 5 вершинами и 7 рёбрами. (b) Представление этого графа с помощью списков смежных вершин. (c) Представление этого графа в виде матрицы смежности.

Рисунок 23.2 23.2 Два представления ориентированного графа. (a) Ориентированный граф G с 6 вершинами и 8 рёбрами. (b) Представление этого графа с помощью списков смежных вершин. (c) Представление этого графа в виде матрицы смежности.

Для ориентированного графа сумма длин всех списков смежных вершин равна общему числу рёбер: ребру (u, v) соответствует элемент v списка $Adj[u]$. Для неориентированного графа эта сумма равна удвоенному числу рёбер, так как ребро (u, v) порождает элемент в списке смежных вершин как для вершины u, так и для v. В обоих случаях количество требуемой памяти есть $O(max(V, E)) = O(V + E)$.

Списки смежных вершин удобны для хранения *графов с весами* (weighted graphs), в которых каждому ребру приписан некоторый вещественный *вес* (weight), то есть задана *весовая функция* (weight function) $w : E \to \mathbb{R}$. В этом случае удобно хранить вес $w(u, v)$ ребра $(u, v) \in E$ вместе с вершиной v в списке вершин, смежных с u. Подобным образом можно хранить и другую информацию, связанную с графом.

Недостаток этого представления таков: если мы хотим узнать, есть ли в графе ребро из u в v, приходится просматривать весь список $Adj[u]$ в поисках v. Этого можно избежать, представив граф в виде матрицы смежности — но тогда потребуется больше памяти.

При использовании *матрицы смежности* мы нумеруем вершины графа (V, E) числами $1, 2, \ldots, |V|$ и рассматриваем матрицу $A = (a_{ij})$ размера $|V| \times |V|$, для которой

$$a_{ij} = \begin{cases} 1, \text{ если } (i, j) \in E, \\ 0 \text{ в противном случае} \end{cases}$$

На рис. 23.1 (c) и 23.2 (c) показаны матрицы смежности неориентированного и ориентированного графов рис. 23.1 (a) и 23.2 (a) соответственно. Матрица смежности требует $\Theta(V^2)$ памяти независимо от количества ребер в графе.

Для неориентированного графа матрица смежности симметрична относительно главной диагонали (как на рис. 3.1(c)), поскольку (u, v) и (v, u) — это одно и то же ребро. Другими словами, матрица смежности неориентированного графа совпадает со своей *транспонированной* (transpose). (Транспонированием называется переход от матрицы $A = (a_{ij})$ к матрице $A^T = (a^T_{ij})$, для которой $a^T_{ij} = a_{ji}$. Благодаря симметрии достаточно хранить только числа на главной диагонали и выше нее, тем самым мы сокращаем требуемую память почти вдвое.

Как и для списков смежных вершин, хранение весов не составляет проблемы: вес $w(u, v)$ ребра (u, v) можно хранить в матрице на пересечении u-той строки и v-го столбца. Для отсутствующих рёбер можно записать специальное значение NIL (в некоторых задачах вместо этого пишут 0 или ∞).

Для небольших графов, когда места в памяти достаточно, матрица смежности бывает удобнее — с ней часто проще работать. Кроме того, если не надо хранить веса, то элементы метрицы смежности представляют собой биты, и их можно размещать по нескольку в одном машинном слове, что даёт заметную экономию памяти.

Упражнения

23.1-1

Граф хранится в виде списков смежности. Сколько операций нужно, чтобы найти число выходящих из данной вершины рёбер? число входящих в данную вершину рёбер?

23.1-2

Укажите представление в виде списков смежности и матрицу смежности для графа, являющегося полным двоичным деревом с 7 вершинами (пронумерованными от 1 до 7 как при сортировке с помощью кучи в главе 7).

23.1-3

Что произойдёт с матрицей смежности ориентированного графа, если обратить направления стрелок на всех его рёбрах, заменив каждое ребро (u, v) на ребро (v, u)? Как обратить напраления стрелок, если граф хранится в форме списков смежности? Оцените требуемое для этого число операций.

23.1-4

Мультиграф $G = (V, E)$ представлен в виде списков смежных вершин. Как за время $O(V + E)$ преобразовать его в обычный неориентированный граф $G' = (V, E')$, заменив кратные рёбра на обычные и удалив рёбра-циклы?

23.1-5

Квадратом (square) ориентированного графа $G = (V, E)$ называется граф $G^2 = (V, E^2)$, построенный так: $(u, w) \in E^2$, если существует вершина $v \in V$, для которой $(u, v) \in E$ и $(v, w) \in E$ (две вершины соединяются ребром, если раньше был путь из двух рёбер). Как преобразовать G в G^2, если граф хранится в виде списков смежных вершин? как матрица смежности? Оцените время работы ваших алгоритмов.

23.1-6

Почти любой алгоритм, использующий матрицы смежности, требует времени $\Theta(V^2)$ (просто на чтение этой матрицы), но бывают и исключения. Покажите, что за время $O(V)$ по матрице смежности можно выяснить, содержит ли ориентированный граф *"сток"* (sink) — вершину, в которую ведут рёбра из всех других вершин и из которой не выходит ни одного ребра.

23.1-7

Назовем *матрицей инцидентности* (incidence matrix) ориентированного графа $G = (E, V)$ матрицу $B = (b_{ij})$ размера $|V| \times |E|$, в которой

$$b_{ij} = \begin{cases} -1, & \text{если ребро } j \text{ выходит из вершины } i, \\ 1, & \text{если ребро } j \text{ входит в вершину } i, \\ 0 & \text{в остальных случаях.} \end{cases}$$

Каков смысл элементов матрицы BB^T? (Здесь B^T — транспонированная матрица.)

23.1.2 Поиск в ширину

Поиск в ширину (breadth-first search) — один из базисных алгоритмов, составляющий основу многих других. Например, алгоритм Дейкстры поиска кратчайших путей из одной вершины (глава 25) и алгоритм Прима поиска минимального покрывающего дерева (раздел 24.2) могут рассматриваться как обобщения поиска в ширину.

Пусть задан граф $G = (V, E)$ и фиксирована *начальная вершина* (source vertex) s. Алгоритм поиска в ширину перечислит все достижимые из s (если идти по рёбрам) вершины в порядке возрастания расстояния от s. Расстоянием считается длина минимального пути из начальной вершины. В процессе поиска из графа выделяется часть, называемая *"деревом поиска в ширину"* с корнем s. Она содержит все достижимые из s вершины (и только их). Для каждой из них путь из корня в дереве поиска будет одним из кратчайших путей (из начальной вершины) в графе. Алгоритм применим и к ориентированным, и к неориентированным графам.

Название объясняется тем, что в процессе поиска мы идём вширь, а не вглубь (сначала просматриваем все соседние вершины, затем соседей соседей и т.д.).

Для наглядности мы будем считать, что в процессе работы алгоритма вершины графа могут быть белыми, серыми и чёрными. Вначале они все белые, но в ходе работы алгоритма белая вершина может стать серой, а серая — чёрной (но не наоборот). Повстречав новую вершину, алгоритм поиска красит её, так что окрашенные (серые или чёрные) вершины — это в точности те, которые уже обнаружены. Различие между серыми и чёрными вершинами используется алгоритмом для управления порядком обхода: серые вершины образуют *"линию фронта"*, а чёрные — *"тыл"*. Более точно, поддерживается такое свойство: если $(u, v) \in E$ и u чёрная, то v — серая или чёрная вершина. Таким образом, только серые вершины могут иметь смежные необнаруженные вершины.

Вначале дерево поиска состоит только из корня — начальной вершины s. Как только алгоритм обнаруживает новую белую вершину v, смежную с ранее найденной вершиной u, вершина v (вместе с ребром (u, v)) добавляется к дереву поиска, становясь *ребёнком* (child) вершины u, а u становится *родителем* (parent) v. Каждая вершина обнаруживается только однажды, так что двух родителей у неё быть не может. Понятия *предка* (ancestor) и *потомка* (descendant) определяются как обычно (потомки — это дети, дети детей, и т.д.). Двигаясь от вершины к корню, мы проходим всех её предков.

Приведенная ниже процедура BFS (breadth-first search — поиск в ширину) использует представление графа $G = (V, E)$ списками смежных вершин. Для каждой вершины u графа дополнительно хранятся её цвет $color[u]$ и её предшественник $\pi[u]$. Если предшественника нет (например, если $u = s$ или u ещё не обнаружена), $\pi[u] = \textsc{nil}$. Кроме того, расстояние от s до u записывается в массив $d[u]$. Процедура использует также очередь Q (FIFO, раздел 11.1) для хранения множества серых вершин.

```
BFS$(G,s)$

for (для) всех вершин $u\in V[G]-\{s\}$
  do $color[u] \leftarrow$ БЕЛЫЙ
    $d[u] \leftarrow \infty$
    $\pi [u] \leftarrow$ NIL
$color[s] \leftarrow$ СЕРЫЙ
$d[s] \leftarrow 0$
$\pi [s] \leftarrow$ NILL
$Q\leftarrow \{s\}$
while $Q\ne \emptyset$
  do $u\leftarrow head[Q]$
     for (для) всех $v\in Adj[u]$
       do if $color[v]=$ БЕЛЫЙ
            then $color[v]\leftarrow$ СЕРЫЙ
                 $d[v]\leftarrow d[u]+1$
                 $\pi[v]\leftarrow u$
                 Enqueue($Q,v$)
      Dequeue($Q$)
      $color[u]\leftarrow$ ЧЕРНЫЙ
```

На рис.~23.3 приведён пример исполнения процедуры
\textsc{BFS}.

\begin{figure}

\caption{
23.3
Исполнение процедуры \textsc{BFS} для неориентированного графа. Рёбра формируемого дерева показаны серыми. Внутри каждой вершины u указано значение $d[u]$. Показано состояние очереди Q перед каждым повторением цикла в строках 9--18. Рядом с элементами очереди показаны расстояния от корня.
}
\end{figure}

В строках 1--4 все вершины становятся белыми, все значения d бесконечными, и родителем всех вершин объявляется \textsc{nil}. Строки 5--8 красят вершину s в серый цвет и выполняют связанные с этим действия: в строке 6 расстояние $d[s]$ объявляется равным 0, а в строке 7 говорится, что родителя у s нет. Наконец, в строке 8 вершина s помещается в очередь Q, и с этого момента очередь будет содержать все серые вершины и только их.

Основной цикл программы (строки 9--18) выполняется, пока очередь непуста, то есть существуют серые вершины (вершины, которые уже обнаружены, но списки смежности которых еще не просмотрены). В строке 10 первая такая вершина помещается в u. Цикл \textbf{for} в строках 11--16 просматривает все смежные с ней вершины. Обнаружив среди них белую вершину, мы делаем её серой (строка 13), объявляем u её родителем (строка 15) и устанавливаем расстояние равным $d[u]+1$ (строка 14). Наконец, эта вершина добавляется в хвост очереди Q (строка 16). После этого уже можно удалить вершину u из очереди Q, перекрасив эту вершину в чёрный цвет (строки 17--18).

Анализ

Начнём с более простого --- оценим время работы описанной процедуры. В процессе работы вершины только темнеют, так что каждая вершина кладётся в очередь не более одного раза (благодаря проверке в строке 12). Следовательно, и вынуть её можно только один раз. Каждая операция с очередью требует $O(1)$ шагов, так что всего на операции с очередью уходит время $O(V)$. Теперь заметим, что список смежных вершин просматривается, лишь когда вершина извлекается из очереди, то есть не более одного раза. Сумма длин всех этих списков равна $|E|$, и всего на их обработку уйдет время $O(E)$. Инициализация требует $O(V)$ шагов, так что всего получается $O(V+E)$. Тем самым время работы процедуры \textsc{BFS} пропорционально размеру представления графа G в виде списков смежных вершин.

Кратчайшие пути.

Как мы говорили, поиск в ширину находит расстояния от начальной вершины s до каждой из достижимых вершин графа $G=(V,E)$. Под расстоянием мы понимаем \emph{длину кратчайшего пути} (shortest-path distance): $\delta (s,v)$ определяется как минимальная длина пути, ведущего из s в v. (Длина пути --- это число рёбер в нём.) Если путей нет вообще, расстояние бесконечно. Пути длины $\delta (s,v)$ из s в v называются \emph{кратчайшими путями}; их может быть несколько. (В главах 25 и 26 мы рассмотрим более общее понятие кратчайшего пути, учитывающее веса рёбер: длина пути есть сумма весов. Сейчас все веса равны единице, так что длина есть число рёбер.)

Докажем несколько свойств определённого таким способом расстояния.

Лемма 23.1

Пусть s --- произвольная вершина графа (ориентированного или нет), а (u,v) --- его ребро. Тогда $$ \delta (s,v)\le \delta (s,u)+1.$$

Доказательство.

Если u достижима за k шагов из s, то и v достижима не более чем за $k+1$ шагов (пройдём по ребру (u,v)), поэтому неравенство выполнено. Если же u недостижима из s, то $\delta (s,u)=\infty$ и неравенство тривиально.

% конец доказательства

% далее текст переписан, и даже формулировки лемм изменены
% (с сохранением из общего числа для последующей нумерации
% - невозможно было оставлять такое длинное и запутанное
% рассуждение по такому простому поводу...

Лемма 23.2

Если $\delta($s$,$v$)>0$, то существует вершина u, до которой расстояние на единицу меньше ($\delta(s,v)=\delta(s,u)+1$) и для которой v является смежной вершиной.

Доказательство. Рассмотрим кратчайший путь из s в v. Он будет иметь длину $\delta(s,v)$. Возьмём вершину u, лежащую на этом пути непосредственно перед v. Нам надо убедиться, что до неё расстояние на единицу меньше. В самом деле, у нас есть ведущий в неё путь длины $\delta(s,v)-1$ (отбросим последнее ребро), а более короткого пути быть не может по предыдущей лемме.

% конец доказательства

В условиях леммы для нахождения кратчайшего пути из s в v достаточно найти кратчайший путь из s в u и добавить к нему ребро (u,v).

\medskip

Теперь мы можем доказать, что поиск в ширину правильно вычисляет длины кратчайших путей.

Работа процедуры \textsc{BFS} делится на начальный этап (строки 1--8) и повторения цикла в строках 9--18. Нас будет интересовать состояние переменных после нескольких таких повторений.

Лемма 23.3

Для всякого целого неотрицательного k существует момент после нескольких повторений тела цикла (строки 10--18), когда выполнены следующие утверждения:

\begin{itemize}
\item
вершины, для которых расстояние от начальной меньше k --- чёрные,
равно k --- серые, больше k --- белые;
\item
в очереди Q находятся серые вершины и только они;
\item
в массиве d хранятся правильные значения расстояния от начальной
вершины для чёрных и серых вершин, и бесконечные значения для
белых;
\item
если v --- серая или чёрная вершина, то
$\delta(s,\pi(v))=\delta(s,v)-1$ и в графе есть ребро
$(v,\pi[v])$; для белых вершин значение π есть \textsc{nil}.
\end{itemize}

Доказательство.

Индукция по k.

После выполнения строк 1--8 все пункты леммы выполнены для $k=0$: на расстоянии k находится единственная вершина (начальная), она серая, остальные белые, серая вершина лежит в очереди, для белых вершин d бесконечно и π равно \textsc{nil}, а для серой вершины значения d и π правильны.

Пусть теперь для некоторого k утверждение леммы выполнено, и после нескольких итераций цикла всё так, как написано в лемме. Что будет происходить после этого? Из очереди будут забираться лежащие в ней вершины, которые мы будем называть \textit{просматриваемыми}. Для смежных с ними белых вершин будут выполняться строки 13--16; в строке 16 они добавляются в конец очереди, и потому мы будем называть их \emph{добавляемыми}. В какой-то момент очереди будут изъяты все находившиеся там изначально вершины, то есть все вершины, находящиеся на расстоянии k, и останутся только вновь добавленные.

(Обратите внимание, что здесь существенно используется правило работы очереди: первым пришёл --- первым ушёл.) В этот момент мы мысленно прервём выполнение процедуры и убедимся, что выполнены все условия леммы для на единицу большего значения k.

Просматриваемые вершины --- это вершины, которые были в очереди; по предположению они находятся на расстоянии k.

Добавляемые вершины находятся на расстоянии $k+1$. В самом деле, они являются смежными с просматриваемыми вершинами, находящимися на расстоянии k, и потому по лемме 23.1 расстояние будет не больше $k+1$. С другой стороны, добавляются только белые вершины, и потому по предположению индукции расстояние до них больше k.

Будут добавлены все вершины, находящиеся на расстоянии $k+1$. В самом деле, если вершина v находится на расстоянии $k+1$, то по лемме 23.2 существует вершина u на расстоянии k, для которой она смежная. Вершина u должна быть среди просматриваемых, и во время её обработки вершина v будет добавлена. Надо только иметь в виду, что вершин u может быть несколько --- но это не мешает делу, так как во время просмотра первой из них вершина v будет добавлена (после чего она будет серой и условие в строке 12 будет ложно, так что второй раз её не добавят).

Поэтому в конце рассматривамого нами этапа в очереди будут все вершины, находящиеся на расстоянии $k+1$. Поскольку при добавлении они делаются серыми, а после просмотра вершина делается чёрной, то условие о цветах будет выполнено.

Строки 14 и 15 обеспечивают выполнение пунктов леммы о расстоянии и о массиве π, что завершает доказательство.

Теперь для доказательства правильности процедуры \textsc{BFS} достаточно взять большое k (больше максимального расстояния от начальной вершины до всех вершин графа). Тогда вершин на расстоянии k (серых, они же вершины в очереди) не будет, алгоритм завершит работу и все массивы будут заполнены правильно. Мы доказали такую теорему:

Теорема 23.4 (Алгоритм \textsc{BFS} правилен)

Работая на графе $G=(V,E)$ (ориентированном или неориентированном) с начальной вершиной s, процедура \textsc{BFS} обнаружит (сделает чёрными) все достижимые из s вершины, и для всех $v\in V$ будет выполнено равенство $d[v]=\delta (s,v)$. Кроме того, для любой вершины $v\ne s$, достижимой из s, один из кратчайших путей из s в v можно получить добавлением ребра $(\pi [v],v)$ к (любому) кратчайшему пути из s в $\pi [v]$. Для недостижимой из s вершины значение $\pi[s]$ равно \textsc{nil}.

% конец доказательства

Деревья поиска в ширину.

В ходе работы процедуры \textsc{BFS} выделяется некоторый подграф --- дерево поиска в ширину, задаваемое полями $\pi[v]$. Более формально, применим процедуру \textsc{BFS} к графу $G=(V,E)$ с начальной вершиной s. Рассмотрим подграф, вершинами которого являются достижимые из s вершины, а рёбрами являются рёбра $(\pi[v],v)$ для всех достижимых v, кроме s.

Лемма 23.5.

Построенный таким образом подграф графа G представляет собой дерево, в котором для каждой вершины v имеется единственный простой путь из s в v. Этот путь будет кратчайшим путём из s в v в графе G.

Доказательство.

Существование пути из s в v (как и то, что он будет кратчайшим) следует из теоремы 23.4 (индукция по расстоянию от s до v). Поэтому граф связен. Поскольку число рёбер в нём на единицу меньше числа вершин, то он

```
является деревом (теорема 5.2).

% конец доказательства

Это дерево называется \emph{подграфом предшествования}
(predecessor subgraph), а также \emph{деревом поиска в ширину}
(breadth-first tree) для данного графа и данной начальной
вершины. (Заметим, что построенное дерево зависит от того, в
каком порядке просматриваются вершины в списках смеж-
ных вершин.)

Если значения в массиве $\pi$ уже вычислены с помощью
процедуры \textsc{BFS}, то крайтчайшие пути из $s$ легко найти:
их печатает процедура \textsc{Print-Path}

\begin{verbatim}

Print-Path$(G,s,v)$

1 if $v=s$
2    then напечатать $s$
3    else if $\pi[v]=NIL$
4            then напечатать "пути из $s$ в $v$ нет"
5            else Print-Path$(G,s,\pi[v])$
6                 напечатать $v$
```

Время выполнения пропорционально длине печатаемого пути (каждый рекурсивный вызов уменьшает расстояние от s на единицу).

Упражнения

23.2-1

Что даст поиск в ширину для ориентированного графа рис. 23.2 (а) и вершины 3 в качестве начальной?

23.2-2

Что даст поиск в ширину для неориентированного графа рис. 23.3 и вершины u в качестве начальной?

23.2-3

Сколько времени будет работать процедура BFS, если граф представлен в виде матрицы смежности (и процедура соответственно изменена)?

23.2-4

Покажите, что при поиске в ширину значение $d[u]$, даваемое алгоритмом, не изменится, если переставить элементы в каждом списке смежных вершин.

23.2-5

Приведите пример ориентированного графа $G = (V, E)$, начальной вершины $s \in V$ и некоторого множества ребер $E_\pi \subseteq E$, для которых для каждой достижимой из s вершины $v \in V$ существует единственный путь из s в v, проходящий по рёбрам из E_π, и этот путь является кратчайшим в G, но множество ребер E_π не совпадает с деревом поиска в ширину, даваемым с помощью процедуры BFS, как ни переставляй элементы в каждом из списков смежных вершин.

23.2-6

Придумайте эффективный алгоритм, выясняющий, является ли данный неориентированный граф двудольным.

23.2-7*

Диаметр (diameter) дерева $T = (V, E)$ определяется как

$$\max_{u,v \in V} \delta(u, v)$$

(то есть как максимальная длина кратчайшего пути между двумя вершинами). Придумайте эффективный алгоритм вычисления диаметра дерева, и оцените время его работы.

23.2-8

Пусть $G = (V, E)$ – неориентированный граф. Придумайте алгоритм, отыскивающий за время $O(V + E)$ путь в графе, который проходит каждое ребро ровно по одному разу в каждую сторону. Как найти выход из лабиринта, имея с собой большой запас одинаковых монет?

23.1.3 Поиск в глубину

Стратегия поиска в глубину такова: идти *"вглубь"*, пока это возможно (есть непройденные рёбра), и возвращаться и искать другой путь, когда таких рёбер нет. Так делается, пока не обнаружены все вершины, достижимые из исходной. Если после этого остаются необнаруженные вершины, можно выбрать одну из них и повторять процесс, и делать так до тех пор, пока мы не обнаружим все вершины графа.

Как и при поиске в ширину, обнаружив (впервые) вершину v, смежную с u, мы отмечаем это событие, помещая в поле $\pi[v]$ значение u. Получается дерево — или несколько деревьев, если поиск повторяется из нескольких вершин. Говоря дальше о поиске в глубину, мы всегда предполагаем, что так и делается (поиск повторяется). Мы получаем *подграф предшествования* (predecessor subgraph), определённый так: $G_\pi = (V, E_\pi)$, где

$$E_\pi = \{(\pi[v], v) : v \in V \text{and} \pi[v] \neq NIL\}$$

Подграф предшествования представляет собой *лес поиска в глубину* (depth-first forest), состоящий из *деревьев поиска в глубину* (deep-first trees).

Алгоритм поиска в глубину также использует цвета вершин. Каждая из вершин вначале белая. Будучи *обнаруженной* (discovered), она становится серой; она станет чёрной, когда будет полностью обработана, то есть когда список смежных с ней вершин будет просмотрен. Каждая вершина попадает ровно в одно дерево поиска в глубину, так что эти деревья не пересекаются.

Помимо этого, поиск в глубину ставит на вершинах *метки времени* (timestamps). Каждая вершина имеет две метки: в $d[v]$ записано, когда эта вершина была обнаружена (и сделана серой), а в $f[v]$ — когда была закончена обработка списка смежных с v вершин (и v стала чёрной).

Эти метки времени используются во многих алгоритмах на графах и полезны для анализа свойств поиска в глубину.

В приводимой далее процедуре DFS (Depth-First Search — поиск в глубину) метки времени $d[v]$ и $f[v]$ являются целыми числами от 1 до $2|V|$; для любой вершины u выполнено неравенство

$$d[u] < f[u] \tag{23.1}$$

Вершина u будет БЕЛОЙ до момента $d[u]$, СЕРОЙ между $d[u]$ и $f[u]$ и ЧЁРНОЙ после $f[u]$.

Исходный граф может быть ориентированным или неориентированным. Переменная *time* — глобальная переменная текущего времени, используемого для пометок.

```
DFS$(G)$

for (для) всех вершин $u\in V[G]$
\hspace{1cm} do $color[u]\leftarrow$ белый
\hspace{2cm} $\pi[u]\leftarrow NIL$
$time\leftarrow 0$
for (для) всех вершин  $u\in V[G]$
\hspace{1cm} do if $color[u]=$ белый
\hspace{2cm} then DFS-Visit$(u)$

DFS-Visit(u)
$color[u]\leftarrow$ белый // вершина $u$ была белой
$d[u]\leftarrow time\leftarrow time+1$
\textsc{for} (для) всех $v\in Adj[u]$ // обработать ребро $(u,v)$
\hspace{1cm} \textsc{do if} $color[v]=$ белый
\hspace{2cm} \textsc{then} $\pi[v]\leftarrow u$
\hspace{3cm} DFS-Visit$(v)$
```

Рисунок 23.3 23.4 Исполнение алгоритма DFS для ориентированного графа. После просмотра каждое ребро становится либо серым (если оно включается в дерево поиска) или пунктирным (обратные рёбра помечены буквой B (back), перекрёстные — буквой C (cross), прямые — буквой F (forward)). У каждой вершины показаны времена начала и конца обработки.

```
7 $color[u]\leftarrow$ чёрный //вершина обработана, делаем её черной
8 $f[u]\leftarrow time\leftarrow time+1$
```

На рис. 23.4 показана работа процедуры DFS на графе рис. 23.2.

В строках 1-3 все вершины красятся в белый цвет; в поле π помещается Nil. В строке 4 уставливается начальное (нулевое) время. В строках 5–7 вызывается процедура DFS-Visit для всех вершин (которые остались белыми к моменту вызова — предыдущие вызовы процедуры могли сделать их чёрными). Эти вершины становятся корнями деревьев поиска в глубину.

В момент вызова DFS-Visit(u) вершина u — белая. В строке 1 она становится серой. В строке 2 время её обнаружения заносится в $d[u]$ (до этого счётчик времени увеличивается на 1). В строках 3-7 просматриваются смежные с u вершины; процедура DFS-Visit вызывается для тех из них, которые оказываются белыми к моменту вызова. После просмотра всех смежных с u вершин мы делаем вершину u чёрной и записываем в $f[u]$ время этого события.

Подсчитаем общее число операций при выполнении процедуры DFS. Циклы в строках 1–3 и 5–7 требуют $\Theta(V)$ времени (помимо вызовов DFS-Visit). Процедура DFS-Visit вызывается ровно один раз для каждой вершины (ей передаётся белая вершина, и она сразу же делает её серой). Во время выполнения DFS-Visit(v) цикл в строках 3–6 выполняется $|Adj[v]|$ раз. Поскольку

$$\sum_{v\in V} |Adj[v]| = \Theta(E),$$

время выполнения строк 3–6 процедуры DFS-Visit составляет $\Theta(E)$. В сумме получается время $\Theta(V+E)$.

Свойства поиска в глубину.

Прежде всего отметим, что подграф предшествования (составленный из деревьев поиска в глубину) в точности соответствует структуре рекурсивных вызовов процедуры DFS-Visit. Именно, $u = \pi[v]$ тогда и только тогда, когда произошёл вызов DFS-Visit(v) во время просмотра списка смежных с u вершин.

Другое важное свойство состоит в том, что времена обнаружения и окончания обработки образуют *правильную скобочную структуру* (parenthesis structure). Обозначая обнаружение вершины u открывающийся скобкой с пометкой u, а окончание её обработки — закрывающейся с той же пометкой, то перечень событий будет

Рисунок 23.4 23.5 Свойства поиска в глубину. (a) Результат поиска (в обозначениях рис. 23.4). (b) Промежутки между обнаружением и окончанием обработки для каждой из вершин, изображённые в виде прямоугольников. Стрелки указывают структуру деревьев посика в глубину. (c) Отмечены рёбра исходного графа с указанием их типов (рёбра дерева и прямые рёбра ведут вниз, обратные ведут вверх).

правильно построенным выражением в том смысле (скобки с одинаковыми пометками считаем парными). Например, поиску в глубину на рис. 23.5 (a) соответствует расстановка скобок, изображённая на рисунке 23.5(b).

Чтобы доказать эти и другие свойства, мы должны рассуждать по индукции. При этом, доказывая требуемое свойство рекурсивной процедуры DFS-Visit, мы предполагаем, что рекурсивные вызовы этой процедуры обладают выбранным свойством.

Убедимся вначале, что вызов DFS-Visit(u) для белой вершины u делает чёрной эту вершину и все белые вершины, доступные из неё по белым путям (в которых все промежуточные вершины также белые), и оставляет серые и чёрные вершины без изменений.

В самом деле, рекурсивные вызовы в строке 6 выполняются лишь для белых вершин, являющихся смежными с u. Если какая-то вершина w была закрашена в ходе этих вызовов, то (по индуктивному предположению) она была доступна по белому пути из одной из белых вершин, смежных с u, и потому доступна по белому пути из u.

Напротив, если вершина w доступна по белому пути из u, то этот она доступна по белому пути из какой-то белой вершины v, смежной с u (посмотрим на первый шаг пути). Будем считать, что v — *первая* из таких вершин (в порядке просмотра в строке 3). В этом случае все вершины белого пути из v в w останутся белыми к моменту вызова DFS-Visit(v), поскольку они недоступны по белым путям из предшествующих v вершин (иначе w была бы также доступна). По индуктивному предположению w станет чёрной после вызова DFS-Visit(v).

Кроме того, сама вершина u станет сначала серой, а потом чёрной. (Заметим, что мы могли бы сразу сделать её чёрной, поскольку программа никак не различает серые и чёрные вершины, однако это различие нам пригодится в дальнейшем.)

Ясно также, что цвета серых и чёрных вершин остаются без изменений (поскольку это верно для рекурсивных вызовов по индуктивному предположению).

Аналогичные рассуждения по индукции позволяют установить, что вызов DFS-Visit(u) меняет поля $\pi[v]$ для всех окрашиваемых вершин v, отличных от u, тем самым формируя из них дерево с корнем в u, а также добавляет к описанному выше протоколу из скобок с пометками правильное скобочное выражение, внешние скобки которого имеют пометку u, а внутри находятся скобки с пометками, соответствующими окрашиваемым вершинам.

Подводя итоги, можно сформулировать такие утверждения:

Теорема 23.6 (о скобочной структуре).

При поиске в глубину в ориентированном или неориентированном графе $G = (V, E)$ для любых двух вершин u и v выполняется ровно одно из следующих трёх утверждений:

отрезки $[d[u], f[u]]$ и $[d[v], f[v]]$ не пересекаются;

отрезок $[d[u], f[u]]$ целиком содержится внутри отрезка $[d[v], f[v]]$ и u — потомок v в дереве поиска в глубину;

отрезок $[d[v], f[v]]$ целиком содержится внутри отрезка $[d[u], f[u]]$ и v — потомок u в дереве поиска в глубину;

Следствие 23.7 (вложение интервалов для потомков)

Вершина v является (отличным от u) потомком вершины u в лесе поиска в глубину для (ориентированного или неориентированного) графа G, если и только если $d[u] < d[v] < f[v] < f[u]$

Теорема 23.8 (о белом пути)

Вершина v является потомком вершины u в лесе поиска в глубину (для ориентированного или неориентированного графа $G = (V, E)$) в том и только том случае, если в момент времени $d[u]$, когда вершина u обнаружена, существует путь из u в v, состоящий только из белых вершин.

Классификация рёбер.

Рёбра графа делятся на несколько категорий в зависимости от их роли при поиске с глубину. Эта классификация оказывается полезной в различных задачах. Например, как мы увидим в следующем разделе, ориентированный граф не имеет циклов тогда и только тогда, когда поиск в глубину не находит в нём *"обратных"* ребер (лемма 23.10).

Итак, пусть мы провели поиск в глубину на графе G и получили лес G_π.

1. *Рёбра дерева* (tree edges) — это рёбра графа G_π. (Ребро (u, v) будет ребром дерева, если вершина v была обнаружена при обработке этого ребра.)

2. *Обратные рёбра* (back edges) — это ребра (u, v), соединяющие вершину u со её предком v в дереве поиска в глубину. (Рёбра-циклы, возможные в ориентированных графах, считаются обратными рёбрами.)

3. *Прямые рёбра* (forward edges) соединяют вершину с её потомком, но не входят в дерево поиска в глубину.

4. *Перекрёстные рёбра* (cross edges) — все остальные рёбра графа. Они могут соединять две вершины одного дерева поиска в глубину, если ни одна из этих вершин не является предком другой, или же вершины из разных деревьев.

На рисунках 23.4 и 23.5 рёбра помечены в соответствии со своим типом. Рис. 23.5(c) показывает граф рис. 23.5(a), нарисованный так, чтобы прямые рёбра и рёбра деревьев вели вниз, а обратные – вверх.

Алгоритм DFS может быть дополнен классификацией рёбер по их типам. Идея здесь в том, что тип ребра (u, v) можно определить по цвету вершины v в тот момент, когда ребро первый раз исследуется (правда, прямые и перекрестные ребра при этом не различаются): белый цвет означает ребро дерева, серый — обратное ребро, чёрный — прямое или перекрёстное ребро.

Чтобы убедиться в этом, надо заметить, что к моменту вызова DFS-Visit(v) серыми являются вершины на пути от корня дерева к вершине v и только они (индукция по глубине вложенности вызова). Чтобы отличить прямые рёбра от перекрёстных, можно воспользоваться полем d: ребро (u, v) оказывается прямым, если $d[u] < d[v]$, и перекрёстным, если $d[u] > d[v]$ (упр. 23.3-4).

Неориентированный требует особого рассмотрения, так как одно и то же ребро $(u, v) = (v, u)$ обрабатываеся дважды, с друх концов, и может попасть в разные категории. Мы будем относить его в той категории, которая стоит раньше в нашем перечне четырёх категорий. Тот же самый результат получится, если мы будем считать, что тип ребра определяется при его первой обработке и не меняется при второй.

Оказывается, что при таких соглашениях прямых и перекрёстных ребер в неориентированном графе не будет.

Теорема 23.9

При поиске в глубину в неориентированном графе G любое ребро оказывается либо прямым, либо обратным.

Доказательство.

Пусть (u, v) — произвольное ребро графа G, и пусть, например, $d[u] < d[v]$. Тогда вершина v должна быть обнаружена и обработана прежде, чем закончится обработка вершины u, так как v содержится в списке смежных с u вершин. Если ребро (u, v) первый раз обрабатывается в направлении от u к v, то (u, v) становится ребром дерева. Если же оно первый раз обрабатывается в направлении от v к u, то оно становится обратным ребром (когда оно исследуется, вершина u — серая).

Эта теорема будет не раз использована в следующих разделах.

Упражнения.

23.3-1

Нарисуйте таблицу 3×3, строки и столбцы которой отмечены как белый, серый и чёрный. В каждой клетке (i, j) пометьте, может ли в процессе поиска в глубину на ориентированном графе найтись ребро из вершины цвета i в вершину цвета j, и какого типа может быть такое ребро. Сделайте аналогичную таблицу для неориентированных графов.

23.3-2

Примените алгоритм поиска в глубину для графа рис. 23.6. Счи-

Рисунок 23.5 23.6 Ориентированный граф для упр. 23.3-2 и 23.3-3.

тайте, что цикл FOR в строках 5-7 процедуры DFS перебирает вершины в алфавитном порядке, и что в списках смежных вершин они тоже идут по алфавиту. Найдите время обнаружения и окончания обработки каждой вершины. Укажите типы всех ребер.

23.3-3

Напишите выражение из скобок, соответствующее поиску в глубину в предыдущем упражнении.

23.3-4

Докажите, что ребро (u, v) является

a. ребром дерева или прямым ребром, если и только если $d[u] < d[v] < f[v] < f[u]$;

b. обратным ребром, если и только если $d[v] < d[u] < f[u] < f[v]$;

c. перекрёстным ребром, если и только если $d[v] < f[v] < d[u] < f[u]$.

23.3-5

Покажите что для неориентированных графов всё равно, определяем ли мы его тип как первый из четырёх возможных в перечне, или как его тип при первой обработке.

23.3-6

Постройте контрпример к такой гипотезе: если в ориентированном графе G существует путь из u в v, и если $d[u] < d[v]$ при поиске в глубину на этом графе, то v — потомок u в построенном лесу поиска в глубину.

23.3-7

Модифицируйте алгоритм поиска в глубину так, чтобы он печатал каждое ребро ориентированного графа вместе с его типом. Какие изменения нужны для неориентированного графа?

23.3-8

Объясните, как вершина может оказаться единственной в дереве поиска в глубину, даже если у неё есть как входящие, так и исходящие ребра.

23.3-9

Покажите, что с помощью поиска в глубину можно найти связные компоненты неориентированного графа, и что лес поиска в глубину будет содержать столько же деревьев, сколько есть связных компонент. Для этого измените алгоритм поиска так, чтобы каждой вершине v он присваивал номер от 1 до k (k — число связных компонент), указывающий, в какой связной компоненте находится v.

23.3-10*

Говорят, что ориентированный граф $G = (V, E)$ обладает свойством *единственности пути* (is singly connected), если в нём не существует двух разных простых путей, имеющих общее начало

Переводы называний: undershorts – трусы, pants – штаны, belt – ремень, shirt – рубашка, tie – галстук, jacket – пиджак, socks – носки, shoes – ботинки, watch – часы.

Рисунок 23.6 23.7 (a) Профессор топологически сортирует свою одежду по утрам. Ребро (u, v) означает, что u должно быть надето до v. Рядом с вершинами показаны времена начала и конца обработки при поиске в глубину. (b) Граф топологически отсортирован (вершины расположены в порядке убывания времени окончания обработки). Все рёбра идут слевао направо.

и общий конец. Постройте эффективный алгоритм, определяющий, обладает ли граф этим свойством.

23.1.4 Топологическая сортировка

Пусть имеется оринетированный граф без циклов (directed acyclic graph; это английское название иногда сокращают до “dag”). Задача о *топологической сортировке* (topological sort) этого графа состоит в следующем: надо указать такой линейный порядок на его вершинах, что любое ребро ведет от меньшей вершины к большей (в смысле этого порядка). Очевидно, что если в графе есть циклы, такого порядка не существует. Можно сформулировать задачу о топологической сортировке и так: расположить вершины графа на горизонтальной прямой так, чтобы все рёбра шли слева направо. (Слово *”сортировка”* не должно вводить в заблуждение: эта задача весьма отличается от обычной задачи сортировки, описанной в части II.)

Вот пример ситуации, в которой возникает такая задача. Рассеянный профессор одевается по утрам, причём какие-то вещи обязательно надо надевать до каких-то других (напрример, носки — до башмаков); в других случаях это всё равно (носки и штаны, например). На рис. 23.7 (a) требуемые соотношения показаны в виде ориентированного графа: ребро (u, v) означает, что предмет u должен быть надет до v. Топологическая сортировка этого графа, тем самым, описывает возможный порядок одевания. Один из таких порядков показан на рис. 23.7 (b) (надо одеваться слева направо).

Следующий простой алгоритм топологически сортирует ориентированный ациклический граф.

```
Topological-Sort($G$)

Вызвать DFS($G$), при этом,
завершая обработку вершины (\textsc{DFS-Visit}, строка 8),
  добавлять её в начало списка
вернуть построенный список вершин
```

На рисунке 23.7 (b) показан результат применения такого алгоритма: значения $f[v]$ убывают слева направо.

Топологическая сортировка выполняется за время $\Theta(V+E)$, потому что столько времени занимает поиск в глубину, а добавить каждую из $|V|$ вершин к списку можно за время $O(1)$.

Правильность этого алгоритма доказывается с помощью такой леммы:

Лемма 23.10

Ориентированный граф не имеет циклов тогда и только тогда, когда поиск в глубину не находит в нём обратных рёбер.

Доказательство.

$\Rightarrow$: Обратно ребро соединяет потомка с предком и потому замыкает цикл, образованный рёбрами дерева.

$\Leftarrow$: Пусть в графе имеется цикл c. Докажем, что в этом случае поиск в глубину обязательно найдёт обратное ребро. Среди вершин цикла выберем вершину v, которая будет обнаружена первой, и пусть (u,v) — ведущее в неё ребро цикла. Тогда в момент времени $d[v]$ из v в u ведёт путь из белых вершин. По теореме о белом пути u станет потомком v в лесе поиска в глубину, поэтому (u,v) будет обратным ребром.

Теорема 23.11

Процедура Topological-Sort(G) правильно выполняет топологическую сортировку ориентированного графа G без циклов.

Доказательство.

Нужно доказать, что для любого ребра (u,v) выполнено неравенство $f[v] < f[u]$. В момент обработки этого ребра вершина v не может быть серой (это означало бы, что она является предком u и (u,v) является обратным ребром, что противоречит лемме 23.10). Поэтому v в этот момент должна быть белой или чёрной. Если v — белая, то она становится ребёнком u, так что $f[v] < f[u]$. Если она уже чёрная, то тем более $f[v] < f[u]$.

Упражнения

23.4-1

В каком порядке расположит вершины графа рис. 23.8 алгоритм Topological-Sort? (Порядок просмотра вершин алфавитный.)

23.4-2

Результат работы алгоритма Topological-Sort зависит от порядка просмотра вершин. Покажите, что различные упорядочения списков смежных вершин могут привести к одному и тому же результату.

23.4-3

Придумайте алгоритм, определяющий, имеется ли в данном неориентированном графе $G=(V,E)$ цикл. Можно ли сделать это за время $O(V)$ (с константой, не зависящей от $|E|$)?

23.4-4

Верно ли, что для ориентированного графа с циклами алгоритм Topological-Sort находит порядок, при котором число *"плохих"* рёбер (идущих в неправильном направлении) будет минимально?

Рисунок 23.7 23.9 (a) Ориентированный граф G и его сильно связные компоненты (показаны серым). (b) Транспонированный граф G^T. Показано дерево поиска в глубину, вычисляемое в строке 3 процедуры STRONGLY-CONNECTED-COMPONENTS Ребра дерева обведены серым. Вершины b, c, g, h, являющиеся корнями деревьев поиска в глубину (для графа G^T), выделены чёрным. (c) Ациклический граф, который получится, если стянуть каждую сильно связную компоненту графа G в точку.

23.4-5

Другой способ топологической сортировки состоит в том, чтобы последовательно находить вершины с входящей степенью 0, печатать их и удалять из графа вместе со всеми выходящими из них рёбрами. Как реализовать эту идею, чтобы время выполнения составило $O(V+E)$? Что произойдёт, если в исходном графе есть циклы?

23.5 Сильно связные компоненты.

23.1.5 Сильно связные компоненты

Классическое применение поиска в глубину — задача о разложении графа на сильно связные компоненты. Мы покажем, как это можно сделать, дважды выполнив поиск в глубину.

Многие алгоритмы, работающие на ориентированных графах, начинают с отыскания сильно связных компонент: после этого задача решается отдельно для каждой компоненты, а потом решения комбинируются в соотвествии со связями между компонентами. Эти связи можно представлять в виде так называемого *"графа компонент"* (упр. 23.5-4).

Вспомним (глава 5), что сильно связной компонентой ориентированного графа $G=(V,E)$ называется максимальное множество вершин $U \subset V$ с таким свойством: любые две вершины u и v из U достижимы друг из друга ($u \rightsquigarrow v$ и $v \rightsquigarrow u$).

Пример графа с выделенными сильно связными компонентами показан на рис. 23.9.

Алгоритм поиска сильно связных компонент графа $G=(V,E)$ будет использовать *"транспонированный"* граф $G^T=(V,E^T)$ (упр. 23.1-3), получаемый из исходного обращением стрелок на рёбрах: $E^T=\{(u,v):(v,u)\in E\}$. Такой граф можно построить за время $O(V+E)$ (мы считаем, что исходный и транспонированный графы заданы с помощью списков смежных вершин). Легко понять, что G и G^T имеют одни и те же сильно связные компоненты (поскольку v достижимо из u в G^T, если и только если u достижимо из v в G^T). На рисунке 23.9 (b) показан трезультат транспонирования графа рис. 23.9 (a).

Следующий алгоритм находит сильно связные компоненты ориентированного графа $G=(V,E)$, используя два поиска в глубину

— для G и для G^T; время работы есть $O(V+E)$.

```
Strongly-Connected-Components($G$)
1 с помощью DFS($G$) найти время окончания обработки $f[u]$ для каждой вершины $u$
2 построить $G^T$
3 вызвать DFS($G^T$), при этом в его внешнем цикле перебирать вершины в порядке убывания величны $f[u]$ (вычисленной в строке 1)
4 сильно связными компонентами будут деревья поиска, построенные на шаге 3.
```

С первого взгляда этот алгоритм кажется несколько загадочным. Его анализ мы начнём с двух полезных наблюдений, относящихся к произвольному ориентированному графу.

Лемма 23.12

Если две вершины принадлежат одной сильно связной компоненте, никакой путь между ними не выходит за пределы этой компоненты.

Доказательство.

Пусть вершина w лежит на пути из u и v, и вершины u и v принадлежат одной сильно связной компоненте. Тогда $u \rightsquigarrow w$, $w \rightsquigarrow v$ и $v \rightsquigarrow u$, откуда всё и следует.

Теорема 23.13

В процессе поиска в глубину вершины одной сильно связной компоненты попадают в одно и то же дерево.

Доказательство.

Для произвольной сильно связной компоненты рассмотрим её вершину, обнаруженную первой (обозначим её r). В этот момент все остальные вершины компоненты ещё белые и потому доступны из r по белым путям (ведущие в них пути не выходят за пределы компоненты по лемме 23.12). Поэтому по теореме о белом пути они будут закрашены при вызове DFS-Visit(r) и войдут в то же дерево поиска.

Возвращаясь к анализу алгоритма Strongly-Connected-Components, договоримся, что $d[u]$ и $f[u]$ будут обозначать время обнаружения и время окончания обработки вершины v, найденные в строке 1 алгоритма, а запись $u \rightsquigarrow v$ будет означать существование пути в G (а не в G^T).

Для каждой вершины u графа определим её *предшественника* (forefather) $\varphi(u)$ как ту из вершин w, достижимых из u, для которой обработка была завершена позднее всех:

$$\varphi(u) = \text{такая вершина } w\text{, что } u \rightsquigarrow w \text{ и } f[w] \text{ максимально}$$

При этом вполне может оказаться, что $\varphi(u) = u$.

Так как любая вершина u достижима сама из себя,

$$f[u] \leqslant f[\varphi(u)] \tag{23.2}$$

Докажем теперь, что $\varphi(\varphi(u)) = \varphi(u)$. В самом деле, для любых двух вершин $u, v \in V$

$$u \leadsto v \Rightarrow f[\varphi(v)] \leqslant f[\varphi(u)], \qquad (23.3)$$

потому что $\{w : v \leadsto w\} \subset \{w : u \leadsto w\}$ и предшественник любой вершины имеет максимальное время завершения обработки среди всех достижимых из неё вершин. Поскольку вершина $\varphi(u)$ достижима из u, из формулы (23.3) получаем, что $f[\varphi(\varphi(u))] \leqslant f[\varphi(u)]$. Кроме того, из неравенства (23.2) имеем $f[\varphi(u)] \leqslant f[\varphi(\varphi(u))]$. Следовательно $f[\varphi(\varphi(u))] = f[\varphi(u)]$ и $\varphi(\varphi(u)) = \varphi(u)$, так как две вершины с одним временем завершения обработки.

Как мы увидим, в каждой сильно связной компоненте есть вершина, являющаяся предшественником всех вершин этой компоненты. При поиске в глубину в G эта вершина обнаруживается первой и оказывается обработанной последней (среди вершин этой компоненты). При поиске в G^T она становится корнем дерева поиска в глубину. Давайте докажем эти свойства.

Фиксируем некоторую сильно связную компоненту S. Рассмотрим вершину v этой компоненты, которая обнаруживается (при поиске в глубину) первой, то есть имеет минимальное значение $d[v]$ среди всех $v \in S$. Изучим ситуацию, которая имеет место непосредственно перед вызовом DFS-Visit(V).

1. В этот момент все вершины из S белые. (Если это не так, вершина v не является *первой* обнаруженной вершиной из S.)

2. Все вершины компоненты S достижимы из v по белым путям. (В самом деле, по лемме 23.12 пути не выходят за пределы S.)

3. Ни одна серая вершина не достижима из v. (В самом деле, в момент вызова DFS-Visit(v) серые вершины образуют путь из корня дерева поиска в v, поэтому v достижима из любой серой вершины. Если бы некоторая серая вершина была достижима из v, то она лежала бы в одной компоненте с v, а все такие вершины белые.)

4. Любая белая вершина w, достижимая из v, достижима по белому пути. (В самом деле, на пути из v в w не может быть серых вершин, поэтому все вершины этого пути или белые, иили чёрные. С другой стороны, при поиске в глубину никогда не возникает ребра из чёрной вершины в белую, поэтому на этом пути не может быть чёрных вершин.)

5. Все вершины компоненты S будут закрашены при вызове DFS-Visit(v) и будут потомками v в дереве поиска. (Следует из теоремы о белом пути.)

6. Все вершины, достижимые из v, имеют меньшее время завершения обработки, чем сама v. (В самом деле, для чёрных вершин это очевидно, так как они уже были обработаны к моменту начала обработки v. Для белых это тоже верно, поскольку они будут обработаны в ходе вызова DFS-Visit(v) до окончания обработки v.)

7. Вершина v является собственным предшественником: $\varphi(v) = v$. (Другая формулировка предыдущего утверждения.)

8. Вершина v является предшественником любой вершины u компоненты S. (В самом деле, из u достижимы те же вершины, что из v, и потому вершина с максимальным временем завершения будет той же самой).

Мы видим, что в каждой сильно связной компоненте есть вершина, которая обнаруживается первой, завершает обрабатываться последней и является предшествеником всех вершин этой компоненты.

Таким образом, мы доказали следующие утверждения:

Теорема 23.14

В ориентированном графе $G = (V, E)$ предшественник $\varphi(u)$ любой вершины $u \in V$ оказывается её предком в дереве поиска в глубину.

Следствие 23.15

При любом поиске в глубину на ориентированном графе $G = (V, E)$ вершины u и $\varphi(u)$ лежат в одной сильно связной компоненте для любой $u \in V$.

Теорема 23.16

В ориентированном графе $G = (V, E)$ две вершины лежат в одной сильно связной компоненте тогда и только тогда, когда они имеют общего предшественника при поиске в глубину.

Итак, задача о нахождении сильно связных компонент свелась к задече отыскания предшественников всех вершин графа. Именно для этого используется поиск в глубину в строке 3 алгоритма Strongly-Connected-Components.

Чтобы понять, как это делается, рассмотрим сначала вершину r с максимальным значением $f[v]$ среди всех вершин графа G. (Говоря о значениях $f[v]$, мы имеем в виду значения, вычисленные в строке 1 алгоритма.) Это вершина будет предшественником любой вершины, из которой она достижима (ни одна вершина графа не имеет большего значения $f[v]$). Таким образом, одну сильно связную компоненту мы нашли — это вершины, из которых достижима r. Другими словами, это вершины, достижимые из v в транспонированном графе.

Отбросив все вершины найденной компоненты, возьмём среди оставшихся вершину r' с максимальным значением $f[v]$. Любая оставшаяся вершина u, из которой достижима r', будет иметь r' своим предшественником (ни одна из отброшенных вершин не достижима из u, иначе и r была бы достижима из u и вершина u попала бы в число отброшенных). Таким образом мы найдём вторую сильно связную компоненту — в неё входят те из оставшихся вершин, из которых достижима вершина r' (другими словами, те из оставшихся, которые достижимы из r' в транспонированном графе).

Теперь понятен смысл строки 3 алгоритма Strongly-Connected-Components: поиск в глубину в транспонированном графе поочерёдно *"отслаивает"* сильно связные компоненты. Скажем то же самое более формально:

Теорема 23.17

Алгоритм Strongly-Connected-Components правильно находит сильно связные компоненты ориентированного графа.

Доказательство.

В строке 3 алгоритма происходит вызов алгоритма DFS на транспонированном графе. Этот алгоритм просматривает вершины в порядке убывания параметра $f[v]$, вычисленного в строке 1. При этом строятся деревья поиска, про которые мы хотим доказать, что они будут сильно связными компонентами.

На каждом шаге цикла рассматривается очередная (в порядке убывания параметра f) вершина v. Вершины уже построенных деревьев поиска в этот момент чёрные, а остальные вершины — белые. (Заметим, что при этом всякая вершина, достижимая в графе G^T из чёрной, сама будет чёрной.)

Если очередная вершина u оказывается чёрной, то алгоритм DFS не делает ничего. Если же она белая, то вызов процедуры DFS-Visit(u) в строке 7 алгоритма DFS сделает её и все достижимые из неё в графе G^T вершины чёрными. Мы должны показать, что эти вершины образуют сильно связную компоненту.

Если какая-то белая вершина v достижима из u в графе G^T, то u будет предшественником v, поскольку u достижима из v в G, никакие чёрные вершины не достижимы из v в G и u имеет максимальное значение параметра f среди всех белых вершин. С другой стороны, если белая вершина v не достижима из u в графе G^T, то она не может иметь u своим предшественником. Чёрные вершины также не могут иметь u своим предшественником (их предшественники найдены на предыдущих шагах). Поэтому множество белых вершин, достижимых из u в графе G^T, совпадает с множеством вершин, имеющих u своим предшественником, то есть является сильно связной компонентой.

Упражнения

23.5-1

Как может изменится количество сильно связных компонент графа при добавлении к нему одного ребра?

23.5-2

Применте алгоритм Strongly-Connected-Components к графу рис. 23.6. Найдите времена завершения, вычисляемые в строке 1, и лес, создаваемый строкой 7. Считайте, что цикл в строках 5-7 алгоритма DFS перебирает вершины в алфавитном порядке и что списки смежных вершин также упорядочены по алфамиту.

23.5-3

Профессор решил, что алгоритм поиска сильно связных компонент можно упростить, если использовать исходный (а не транспонированноый) граф при втором поиске, но вершины сортировать в порядке *увеличения* времён завершения. Прав ли он?

23.5-4

Пусть G — ориентированный граф. Если стянуть каждую его сильно связную компоненту в точку, и после этого отождествить рёбра с одинаковыми началами и концами, получится *граф компонент* (component graph) $G^{SCC} = (V^{SCC}, E^{SCC})$. Другими словами, элементами V^{SCC} являются сильно связные компоненты G, и E^{SCC} содержит ребро (u, v) в том и только в том случае, если в G есть ориентированное ребро, начало которого принадлежит u, а конец — v (см. пример на рис. 23.9 (c)). Докажите, что граф компонент не имеет циклов.

23.5-5

Постройте алгоритм, находящий за время $O(E+V)$ граф компонент данного ориентированного графа. (В строимом графе любые две вершины должны быть соединены не более чем одним ребром.)

23.5-6

Дан ориентированный граф $G = (V, E)$. Как построить как можно меньший граф $G' = (V, E')$, который имел бы те же самые сильно связные компоненты и тот же граф компонент? Постройте эффективный алгоритм решения этой задачи.

23.5-7

Ориентированный граф называется $G = (V, E)$ называется *полусвязным* (semiconnected), если для любых двух его вершин u и v либо v достижима из u, либо u достижима из v. Придумайте эффективный алгоритм, определяющий, будет ли граф полусвязным. Каково время его работы?

Задачи

23-1 Классификация ребер при поиске в ширину

Классифицируя рёбра графа по типам, мы исходили из дерева поиска в глубину. Аналогичная классификаия возможно и для дерева поиска в ширину (для рёбер, достижимых из начальной вершины).

a. Докажите следующие свойства для случая поиска в ширину в неориентированном графе:

1. Не бывает прямых и обратных рёбер.

2. Если (u, v) — ребро дерева, то $d[v] = d[u] + 1$.

3. Если (u, v) — перекрёстное ребро, то $d[v] = d[u]$ или $d[v] = d[u] + 1$.

b. Докажите следующие свойства для случая поиска в ширину в ориентированном графе:

1. Не бывает прямых рёбер.

2. Если (u, v) – ребро дерева, то $d[v] = d[u] + 1$.

3. Если (u, v) – перекрёстное ребро, то $d[v] \leqslant d[u] + 1$.

4. Если (u, v) – обратное ребро, то $0 \leqslant d[v] < d[u]$.

Рисунок 23.8 23.10 Точки раздела, мосты и двусвязные компоненты связного неориентированного графа (задача 23-2). Точки раздела и мосты — тёмно-серые, двусвязные компоненты — наборы рёбер в пределах одной серой области (внутри которой указано *bcc*).

23-2 Точки раздела, мосты и двусвязные компоненты

Пусть $G = (V, E)$ – связный неориентированный граф. *Точка раздела* (articulation point) графа G — это вершина, при удалении которой граф G перестаёт быть связным. *Мост* (bridge) — это ребро с аналогичным свойством. *Двусвязная компонента* (biconnected component) — это максимальный набор рёбер, любые два ребра которого принадлежат общему простому циклу (см. пример на рис. 23.10). Точки раздела, мосты и двусвязные компоненты можно найти с помощью поиска в глубину.

Пусть $G_\pi = (V, E_\pi)$ — дерево поиска в глубину в графе G.

a. Докажите, что корень G_π является точкой раздела, если и только если у него более одного сына в G_π.

b. Пусть v — отличная от корня вершина G_π. Докажите, что v — точка раздела G, если и только если не существует обратного ребра (u, w), для которого u — потомок v и w — предок v в G_π, отличный от w.

c. Пусть $low[v]$ — минимальное число среди $d[v]$ и чисел $d[w]$ для всех w, для которых имеется обратное ребро (u, w) для некоторой вершины u, являющейся потомком v. Покажите, как можно вычислить $low[v]$ для всех $v \in V$ за время $O(V)$.

d. Как найти все точки раздела за время $O(E)$?

e. Докажите, что ребро графа G является мостом в том и только том случае, когда оно не входит ни в какой простой цикл.

f. Как найти все мосты графа G за время $O(E)$?

g. Докажите, что двусвязные компоненты графа составляют разбиение множества всех рёбер графа, не являющихся мостами.

h. Придумайте алгоритм, который за время $O(E)$ помечает каждое ребро e графа G некоторым целым числом $bcc[e]$, при этом метки двух рёбер совпадают, если и только если рёбра принадлежат одной двусвязной компоненте.

23-3 Эйлеров цикл

Эйлеровым циклом(Euler tour) связного ориентированного графа $G = (V, E)$ называется цикл, проходящий по каждому ребру G ровно один раз (в одной и той же вершине можно бывать многократно).

a. Покажите, что в G есть эйлеров цикл тогда и только тогда, когда входящая степень каждой вершины равна ее исходящей степени.

b. Придумайте алгоритм, который за время $O(E)$ находит в графе эйлеров цикл (если таковой имеется). (Указание: объединяйте циклы, у которых нет общих рёбер.)

Замечания

Прекрасные руководства по алгоритмам на графах написали Ивен [65] и Тарьян [188].

Поиск в ширину рассмотрел Мур [150] при изучении путей в лабиринтах. Ли [134] независимо открыл тот же алгоритм применительно к соединениям контактов в электронных схемах.

Хопкрофт и Тарьян [102] указали на пользу представления в виде списков смежных вершин для редких графов и обнаружили важность поиска в глубину для построения алгоритмов на графах. Сам по себе поиск в глубину начал использоваться незадолго до 1960 года, в превую очередь в программах, связанных с *"искусственым интеллектом"*.

Тарьян [185] придумал алгоритм поиска сильно связных компонент за линейное время. Алгоритм раздела 23.5 взят из книги Ахо, Хопкрофта и Ульмана [5], которые ссылаются на Косараю (S.R. Kosaraju) и Шарира (M. Sharir). Кнут [121] первым построил алгоритм тополигической сортировки за линейное время.

24

Минимальные покрывающие деревья

Пусть даны n контактов на печатной плате, которые мы хотим электрически соединить. Для этого достаточно использовать $n-1$ проводов, каждый из которых соединяет два контакта. При этом мы обычно стремимся сделать суммарную длину проводов как можно меньше.

Упрощая ситуацию, можно сформулировать задачу так. Пусть имеется связный неориентированный граф $G = (V, E)$, в котором V — множество контактов, а E — множество их возможных попарных соединений. Для каждоого ребра графа (u, v) задан вес $w(u, v)$ (длина провода, необходимого для соединения u и v). Задача состоит в нахождении подмножества $T \subset E$, связывающего все вершины, для которого суммарный вес

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

минимален. Такое подмножество T будет деревом (поскольку не имеет циклов: в любом цикле один из проводов можно удалить, не нарушая связности). Связный подграф графа G, являющийся деревом и содержащий все его вершины, называют *покрывающим деревом* (spanning tree) этого графа. (Иногда используют термин *"остовное дерево"*; для краткости мы будем говорить просто *"остов"*.)

В этой главе мы рассматриваем задачу *о минимальном покрывающем дереве* (minimum-spanning-tree problem). (Здесь слово *"минимальное"* означает *"имеющее минимально возможный вес"*.)

На рисунке 24.1 приведён пример связного графа и его минимального остова.

[Возвращаясь к примеру с проводниками на печатной плате, объясним, почему задача о минимальном дереве является упрощением реальной ситуации. В самом деле, если соединяемые контакты находятся в вершинах единичного квадрата, разрешается соединять его любые вершины и вес соединения равен его длине, то минимальное покрывающее дерево будет состоять из трёх сторон квадрата. Между тем все его четыре вершины можно электрически соединить

Рисунок 24.1 24.1 Минимальное покрывающее дерево. На каждом ребре графа указан вес. Выделены ребра минимального покрывающего дерева (суммарный вес 37). Такое дерево не единственно: заменяя ребро (b, c) ребром (a, h), получаем другое дерево того же веса 37.

двумя пересекающимися диагоналями (суммарная длина $2\sqrt{2} < 3$) и даже ещё короче (введя две промежуточные точки, в которых проводники сходятся под углом 120°.).]

В этой главе мы рассмотрим два способа решения задачи о минимальном покрывающем дереве: алгоритмы Крускала и Прима. Каждый их них легко реализовать с временем работы $O(E \lg V)$, используя обычные двоичные кучи. Применив фибоначчиевы кучи, можно сократить время работы алгоритма Прима до $O(E + V \lg V)$ (что меньше $E + \lg V$, если $|V|$ много меньше $|E|$).

Оба алгоритма (Крускала и Прима) следуют *"жадной"* стратегии: на каждом шаге выбирается *"локально наилучший"* вариант. Не для всех задач такой выбор приведёт к оптимальному решению, но для задачи о покрывающем дереве это так. (Мы обсуждали это в главе 17; задача о покрывающем дереве является превосходной иллюстрацией введённых там понятий.)

В разделе 24.1 описана общая схема алгоритма построения минимального остова (добавление рёбер одного за другим). В разделе 24.2 указаны две конкретных реализации общей схемы. Первый алгоритм, восходящий к Крускалу, аналогичен алгоритму поиска связных компонент из раздела 22.1. Другой (алгоритм Прима) аналогичен алгоритму Дейкстры поиска кратчайших путей (раздел 25.2).

24.1 Построение минимального остова

Итак, пусть дан связный неориентированный графа $G = (V, E)$ и весовая функцией $w : E \to \mathbb{R}$. Мы хотим найти минимальное покрывающее дерево (остов), следуя жадной стратегии.

Общая схема всех наших алгоритмов будет такова. Искомый остов строится постепенно: к изначально пустому множеству A на каждом шаге добавляется одно ребро. Множество A всегда является подмножеством некоторого минимального остова. Ребро (u, v), добавляемое на очередном шаге, выбирается так, чтобы не нарушить этого свойства: $A \cup \{(u, v)\}$ тоже должно быть подмножеством минимального остова. Мы называем такое ребро *безопасным ребром* (safe edge) для A.

```
\textsc{Generic-MST} $(G,w)$
1 $A \leftarrow \emptyset$
2 {\bf while} $A$ не является остовом
```

Рисунок 24.2 24.2 Два изображения одного и того же разреза графа с рисунка 24.1. (a) Вершины множества S изображены белым цветом, его дополнения $V \setminus S$ — черным. Рёбра, пересекающие разрез, соединяют белые вершины с черными. Единственное лёгкое ребро, пересекающее разрез — ребро (d, c). Множество A состоит из выделенных серым цветом выделено. Разрез $(S, V \setminus S)$ согласован с A (ни одно ребро из A не пересекает разрез). (b) Вершины множества S изображены слева, вершины $V \setminus S$ — справа. Ребро пересекает разрез, если оно пересекает вертикальную прямую.

```
3     {\bf do} найти безопасное ребро $(u,v)$ для $A$
4          $A \leftarrow A \cup \{(u,v)\}$
5 {\bf return} $A$
```

По определению безопасного ребра цикл не нарушается свойства *"A является подмножеством некоторого минимального остова"* (для пустого множества это свойство, очевидно, выполнено), так что в строке 5 алгоритм выдаёт минимальный остов. Конечно, главный вопрос в том, как искать безопасное ребро в строке 3. Такое ребро существует (если A является подмножеством минимального остова, то любое ребро этого остова, не входящее в A, является безопасным).

Заметим, что множество A не может содержать циклов (поскольку является частью минимального остова). Поэтому добавляемое в строке 4 ребро соединяет различные компоненты графа $G_A = (V, A)$, и с каждой итерацией цикла число компонент уменьшается на 1. Вначале каждая точка представляет собой отдельную компоненту; в конце весь остов — одна компонента, так что цикл повторяется $|V| - 1$ раз.

В оставшейся части этого раздела мы приведем правило отыскания безопасных ребер (теорема 24.1). В следующем разделе будут описаны два алгоритма, использующих это правило для эффективного поиска безопасных ребер.

Начнём с определений. *Разрезом*(cut) $(S, V \setminus S)$ неориентированного графа $G = (V, E)$ называется разбиение множества его вершин на два подмножества (рис. 24.2).

Говорят, что ребро $(u, v) \in E$ *пересекает* (crosses) разрез $(S, V \setminus S)$, если один из его концов лежит в S, а другой — в $V \setminus S$. Разрез *согласован* с множеством рёбер A (respects the set A), если ни одно ребро из A не пересекает этот разрез. Среди пересекающих разрез рёбер выделяют рёбра наименьшего веса (среди пересекающих этот разрез), называя их *лёгкими* (light edges).

Теорема 24.1

Пусть $G = (V, E)$ — связный неориентированный граф, на множестве вершин которого определена вещественная функция w. Пусть A — множество рёбер, являющееся подмножеством некоторого минимального остова графа G. Пусть $(S, V \setminus S)$ — разрез графа G, согласованный с A, а (u, v) — лёгкое ребро для этого разреза. То-

Рисунок 24.3 24.3 (к доказательству теоремы 24.1). Вершины S — чёрные, вершины $V \setminus S$ — белые. Изображены только рёбра минимального остова (назовём его T). Рёбра множества A выделены серым цветом; (u, v) — лёгкое ребро, пересекающее разрез $(S, V \setminus S)$; (x, y) — ребро единственного пути p от u к v в T.

гда ребро (u, v) является безопасным для A.

Доказательство

Пусть T — минимальный остов, содержащий A. Предположим, что T не содержит ребра (u, v), поскольку в противном случае доказываемое утверждение очевидно. Покажем, что в этом случае существует другой минимальный остов T', содержащий ребро (u, v), так что это ребро является безопасным.

Остов T связен и потому содержит некоторый путь p из u в v (рис. 24.3). Поскольку вершины u и v принадлежат разным частям разреза $(S, V \setminus S)$, в пути p есть по крайней мере одно ребро (x, y), пересекающее разрез. Это ребро не лежит в A, так как разрез согласован с A. Удаление из дерева T ребра (x, y) разбивает его на две компоненты. Добавление (u, v) объединяет эти компоненты в новый остов $T' = T \setminus \{(x, y)\} \cup \{(u, v)\}$.

Покажем, что T' — минимальный остов. Поскольку (u, v) — лёгкое ребро, пересекающее разрез $(S, V \setminus S)$, изъятое из T ребро (x, y) имеет не меньший вес, чем добавленное вместо него ребро (u, v), так что все остова мог только уменьшиться. Но остов был минимальным, значит, вес его остался прежним, и новый остов T' будет другим минимальным остовом (того же веса). Поэтому ребро (u, v), содержащееся в T', является безопасным.

Следующее следствие теоремы 24.1 будет использовано в разделе 24.2.

Следствие 24.2

Пусть $G = (V, E)$ — связный неориентированный граф и на множестве E определена вещественная функция w. Пусть A — множество рёбер графа, являющееся подмножеством некоторого минимального остова. Рассмотрим лес $G_A = (V, A)$. Пусть дерево C — одна из связных компонент леса $G_A = (V, A)$. Рассмотрим все рёбра графа, соединящие вершины из C с вершинами не из C, и возьмём среди них ребро наименьшего веса. Тогда это ребро безопасно для A.

Доказательство

очевидно: разрез $(C, V \setminus C)$ согласован с A, а ребро (u, v) — лёгкое ребро для этого разреза.

Упражнения

24.1-1

Пусть (u, v) — ребро минимального веса в графе G. Покажите, что (u, v) принадлежит некоторому минимальному остову этого графа.

24.1-2

Профессор утверждает, что верно следующее обращение теоремы 24.1. Пусть $G = (V, E)$ — связный неориентированный граф, и на множестве E определена вещественная функция w. Пусть A — подмножество E, являющееся подмножеством некоторого минимального остова G. Пусть $(S, V \setminus S)$ — любой разрез G, согласованный с A, а (u, v) — безопасное ребро для A, пересекающее $(S, V \setminus S)$. Тогда ребро (u, v) является лёгким ребром, пересекающим этот разрез. Постройте контрпример к утверждению профессора.

24.1-3

Покажите, что если ребро (u, v) содержится в некотором минимальном остове, то существует некоторый разрез графа, для которого оно является лёгким ребром, пересекающим этот разрез.

24.1-4

Рассмотрим множество всех рёбер, которые являются лёгкими рёбрами всевозможных разрезов графа. Приведите простой пример, когда это множество не является минимальным остовом.

24.1-5

Пусть e — ребро максимального веса в некотором цикле графа $G = (V, E)$. Докажите, что существует минимальный остов графа $G = (V, E \setminus \{e\})$, который является также минимальным остовом графа G.

24.1-6

Покажите, что если для любого разреза графа существует единственное лёгкое ребро, пересекающее этот разрез, то в графе есть только один минимальный остов. Покажите, что обратное утверждение неверно.

24.1-7

Объясните, почему в графе с положительными весами рёбер любое подмножество рёбер, связывающее все вершины и обладающее минимальным суммарным весом (среди таких подмножеств), является деревом. Приведите пример, показывающий, что это заключение перестаёт быть верным, если веса рёбер могут быть отрицательными.

24.1-8

Пусть T — минимальный остов графа G. Составим упорядоченный список весов всех рёбер остова T. Покажите, что для любого другого минимального остова получится тот же самый список.

24.1-9

Пусть T — минимальный остов графа $G = (V, E)$. Пусть V' — подмножество V. Через T' обозначим подграф T, порождённый V', а через G' — подграф G, порождённый V'. Покажите, что если T' связен, то T' — минимальный остов G'.

24.2 Алгоритмы Крускала и Прима

Оба этих алгоритма следуют описанной схеме, но по-разному выбирают безопасное ребро. В алгоритме Крускала множество рёбер A представляет собой лес, состоящий из нескольких связных компонент (деревьев). Добавляется ребро минимального веса среди всех рёбер, концы которых лежат в разных компонентах.

В алгоритме Приме множество A представляет собой одно дерево, Безопасное ребро, добавляемое к A, выбирается как ребро наименьшего веса, соединяющее это уже построенное дерево с некоторой новой вершиной.

Алгоритм Крускала

В любой момент работы алгоритма Крускала множество A выбранных рёбер (часть будущего остова) не содержит циклов. Оно соединяет вершины графа в несколько связных компонент, каждая из которых является деревом. Среди всех рёре, соединяющих вершины из разных компонент, берётся ребро наименьшего веса. Надо проверить, что оно является безопасным.

Пусть (u, v) — такое ребро, соединяющее вершины из компонент C_1 и C_2. Это ребро является лёгким ребром для разреза $C_1, V \backslash C_1$, и можно воспользоваться теоремой 24.1 (или прямо следствием 24.2).

Реализация алгоритма Крускала напоминает алгоритм вычисления связных компонент (разд. 22.1) и использует структуры данных для непересекающихся множеств (гл. 22), Элементами множеств являются вершины графа. Напомним, что Find-Set(u) возвращает представителя множества, содержащего элемент u. Две вершины u и v принадлежат одному множеству (компоненте), если Find-Set(u) = Find-Set(v). Объединение деревьев выполняется процедурой Union.

```
\textsc{MST-Kruskal}$(G,w)$
1 $A \leftarrow \emptyset$
2 {\bf for} $v\in V[G]$
3    {\bf do} \textsc{Make-Set}$(v)$
4 упорядочить рёбра $E$ по весам
5 {\bf for} $(u,v)\in E$ (в порядке возрастания веса)
6    {\bf do if} \textsc{Find-Set}$(u) \noteq$ \textsc{Find-Set}$(v)$
7          {\bf then} $A \leftarrow A \cup \{(u,v)\}$
8                    \textsc{Union} $(u,v)$
9 {\bf return} $A$
```

На рисунке 24.4 показа работа алгоритма. Сначала (строки 1–3) множество A пусто, и есть $|V|$ деревьев, каждое из которых содержит по одной вершине. В строке 4 рёбра из E упорядочиваются по неубыванию веса. В цикле (строки 5–8) мы проверяем, лежат ли

Рисунок 24.4 24.4 Работа алгоритма Крускала на графе рис. 24.1. Рёбра растущего леса (A) выделены серым цветом. Рёбра рассматриваются в порядке неубывания весов (текущее ребро показано стрелкой). Если ребро соединяет два различных дерева, оно добавляется к лесу, а деревья сливаются.

концы ребра в одном дереве. Если да, то ребро нельзя добавить к лесу (не создавая цикла), и оно отбрасывается. Если нет, то ребро добавляется к A (строка 7), и два соединённых им дерева объединяются в одно (строка 8).

Подсчитаем время работы алгоритма Крускала. Будем считать, что для хранения непересекающихся множеств используется метод раздела 22.3 (с объединением по рангу и сжатием путей — самый быстрый из известных). Инициализация занимает время $O(V)$, упорядочение рёбер в строке 4 — $O(E \lg E)$. Далее производится $O(E)$ операций, в совокупности занимающих время $O(E\alpha(E,V))$, где α — функция, обратная к функции Аккермана (см. раздел 22.4). Поскольку $\alpha(E,V) = o(\lg E)$, общее время работы алгоритма Крускала составляет $O(E \lg E)$ (основное время уходит на сортировку).

Алгоритм Прима

Как и алгоритм Крускала, алгоритм Прима следует общей схеме алгоритма построения минимального остова из раздела 24.1. Он похож на алгоритм Дейкстры поиска кратчайшего пути в графе (раздел 25.2). В этом алгоритме растущая часть остова представляет собой дерево (множество рёбер которого есть A). Как показано на рис. 24.5, формирование дерева начинается с произвольной корневой вершины r. На каждом шаге добавляется ребро наименьшего веса среди рёбер соединяющих вершины этого дерева с вершинами не из дерева. По следствию 24.2 такие рёбра являются безопасными для A, так что в результате получается минимальный остов.

При реализации важно быстро выбирать лёгкое ребро. Алгоритм получает на вход связный граф G и корень r минимального покрывающего дерева. В ходе алгоритма все вершины, ещё не попавшие в дерево, хранятся в очереди с приоритетами. Приоритет вершины v определяется значением $key[v]$, которое равно минимальному весу рёбер, соединяющих v с вершинами дерева A. (Если таких рёбер нет, полагаем $key[v] = \infty$.) Поле $\pi[v]$ для вершин дерева указывает на родителя, а для вершины $v \in Q$ указывает на вершину дерева, в которую ведёт ребро веса $key[v]$ (одно из таких рёбер, если их несколько). Мы не храним множество A вершин строимого дерева явно; его можно восстановить как

$$A = \{(v, \pi[v]) : v \in V \setminus \{r\} \setminus Q\}.$$

В конец работы алгоритма очередь Q пуста, и множество

$$A = \{(v, \pi[v]) : v \in V \setminus \{r\}\}.$$

есть множество вершин покрывающего дерева.

Рисунок 24.5 24.5 Работа алгоритма Прима на графе рис. 24.1 с корневой вершиной a Рёбра, входящие в дерево A. выделены серым; вершины дерева — чёрным. На каждом шаге к A добавляется ребро, пересекающее разрез между деревом и его дополнением. Например, на втором шаге можно было бы добавить любое из рёбер (b, c) и (a, h).

```
\textsc{MST-Prim}$(G,W,r)$
 1 $Q \leftarrow V[G]$
 2 {\bf for} $u\in Q$
 3     {\bf do} $key[u] \gets \infty$
 4 $key[r] \gets 0$
 5 $\pi[r] \gets \textsc{nil}$
 6 {\bf while} $Q \noteq \emptyset$
 7     {\bf do} $u \leftarrow \textsc{Extract-Min}(Q)$
 8         {\bf for} $v\in Adj[u]$
 9             {\bf do if} $v\in Q$ и $w(u,v)<key[v]$
10                 {\bf then} $\pi(v) \leftarrow u$
11                            $key(v) \leftarrow w(u,v)$
```

На рис. 24.5 показана работа алгоритма Прима. После исполнения строк 1–5 и первого прохода цикла в строках 6 – –11 дерево состоит из единственной вершины r, все остальные вершины находятся в очереди, и значение $key[v]$ для них равно длине ребра из r в v или $+\infty$, если такого ребра нет (в первом случае $\pi[v] = r$. Таким образом, выполнен описанный выше инвариант (дерево есть часть некоторого остова, для вершин дерева поле π указывает на родителя, а для остальных вершин на *"ближайшую"* вершину дерева — вес ребра до неё хранится в $key[v]$.

Время работы алгоритма Прима зависит от того, как равлизована очередь Q. Если использовать двоичную кучу (глава 7), инициализацию в строках 1–4 можно выполнить с помощью процедуры Build-Heap за время $O(V)$. Далее цикл выполняется $|V|$ раз, и каждая операция Extract-Min занимает время $O(\lg V)$, всего $O(V \lg V)$. Цикл for в строках 8–11 выполняется в общей сложности $O(E)$ раз, поскольку сумма степеней вершин графа равна $2|E|$. Проверку принадлежности в строке 9 внутри цикла for можно реализовать за время $O(1)$, если хранить состояние очереди ещё и как битовый вектор размера $|V|$. Присваивание в строке 11 подразумевает выполнение операции уменьшения ключа (Decrease-Key), которая для двоичной кучи может быть выполнена за время $O(\lg V)$. Таким образом, всего получаем $O(V \lg V + E \lg V) = O(E \lg V)$ — та же самая оценка, что была для алгоритма Крускала.

Однако эта оценка может быть улучшена, если использовать в алгоритме Прима фибоначчиевы кучи. Как мы видели в главе 21, с помощью фиббоначиевой кучи можно выполнять операцию Extract-Min за учётное время $O(\lg V)$, а операцию Decrease-

Key — за (учётное) время $O(1)$. (Нас интересует именно суммарное время выполнения последовательности операций, так что амортизированный анализ тут в самый раз.) Поэтому при использовании фибоначчиевых куч для реализации очереди время работы алгоритма Прима составит $O(E + V \lg V)$.

Упражнения

24.2-1

Для одного и того же графа G алгоритм Крускала может давать разные результаты (если по-разному упорядочить рёбра одинакового веса). Покажите, что для каждого минимального остова T графа G существует упорядочение рёбер графа G, при котором алгоритм Крускала даст как раз T.

24.2-2

Граф (V, E) задан матрицей смежности. Постройте простую реализацию алгоритма Прима, время работы которой есть $O(V^2)$.

24.2-3

Имеется ли выигрыш при переходе от двоичных куч к фибоначчиевым для разреженного графа $G = (V, E)$ для которого $|E| = \Theta(V)$? для плотного графе, где $|E| = \Theta(V^2)$? При каком соотношении между $|E|$ и $|V|$ переход к фибоначчиевым кучам приводит к асимптотческому улучшению эффективности?

24.2-4

Пусть веса рёбер графа $G = (V, E)$ — целые числа в интервале от 1 до $|V|$. Какой скорости работы алгоритма Крускала можно добиться? А если веса — целые числа от 1 до W (где W — некоторая константа)?

24.2-5

Пусть веса рёбер графа $G = (V, E)$ — целые числа в интервале от 1 до $|V|$. Какой скорости работы алгоритма Прима можно добиться? А если веса — целые числа от 1 до W (где W — некоторая константа)?

24.2-6

Укажите эффективный алгоритм для такой задачи: для данного графа G и весовой функцией w найти наилучшее покрывающее дерево, если критерием *"качества"* дерева считать не сумму весов, а вес самого тяжёлого ребра.

24.2-7*

Пусть известно, что веса рёбер графа — независимые случайные числа, равномерно распределенные на полуинтервале $[0, 1)$. Как это использовать для ускорения работы одного из алгоритмов (Крускала или Прима)?

24.2-8*

Пусть минимальный остов графа G уже построен. Как быстро можно найти новый минимальный остов, если добавить к графу G новую вершину и инцидентные ей рёбра?

Задачи

24-1 Второй по величине остов

Пусть $G = (V, E)$ — неориентированный связный граф с весовой функцией $w : E \to \mathbb{R}$, и пусть $|E| \geqslant |V|$ (число рёбер больше минимально возможного). Упорядочим все покрывающие деревья в порядке неубывания весов; нас будет интересовать второе по величине в этом порядке. (Будем считать для простоты, что все деревья в этом списке имеют различные суммы весов.)

a. Пусть T — минимальное покрывающее дерево графа G. Докажите, что второе по величине дерево получается из T заменой некоторого ребра $(u, v) \in T$ на другое ребро $(x, y) \notin T$.

b. Пусть T — покрывающее дерево графа G. Для любых двух вершин $u, v \in V$ через $max[u, v]$ обозначим ребро максимального веса на единственном пути, соединяющем u и v в T. Укажите алгоритм с временем работы $O(V^2)$, который который для любого заданного T находит $max[u, v]$ для всех пар вершин $u, v \in V$.

c. Укажите эффективный алгоритм поиска второго по величине покрывающего дерева.

24-2 Минимальный остов в разреженном графе

Для очень разреженного связного графа $G = (V, E)$ время работы $O(E + V \lg V)$ алгоритма Прима (с фибоначчиевыми кучами) можно ещё сократить, если предварительно преобразовать граф G, уменьшив число его вершин. Описанная ниже процедура преобразования MST-Reduce получает на вход граф G с весовой функцией и возвращает *"сжатую"* версию G' графа G, одновременно добавляя рёбра к будущему остову. Эта процедура использует массив $orig[u, v]$; в начальный момент $orig[u, v] = (u, v)$.

Идея проста: если для некоторой вершины взять кратчайшее ребро, из неё выходящее, то можно искать минимальный остов среди остовов, включающих это ребро — а эта задача сводится к задаче поиска остова для меньшего графа (с отождествлёнными вершинами). Рёбра, по которым проведена склейка, помещаются в T, а для каждого ребра (u, v), соединяющего вершины нового графа, хранится $w[u, v]$ — то ребро исходного графа, из которого оно произошло (если таких было несколько, то берётся кратчайшее).

```
\textsc{MST-Reduce} (G,T)
{\bf for} $v\in V[G]$
    {\bf do} $mark[v] \leftarrow \textsc{false}$
        \textsc{Make-Set(v)}
{\bf for} $u\in V[G]$
    {\bf do if} $mark[u] = \textsc{false}$
        {\bf then} выбрать $v\in Adj[u]$ с наименьшим $w[u,v]$
                  \textsc{Union(u,v)}
                  $T \leftarrow T\cup \{orig[u,v]\}$
                  $mark[u] \leftarrow $mark[v] \leftarrow
```

```
                                                     \textsc{true}$
$V[G'] \leftarrow \{\textsc{Find-Set(v)} : v\in V[G] \}$
$E[G'] \leftarrow \emptyset$
{\bf for} $(x,y)\in E[G]$
    {\bf do} $u \leftarrow \textsc{Find-Set(x)}$
             $v \leftarrow \textsc{Find-Set(y)}$
             {\bf if} $(u,v)\notin E[G']$
                 {\bf then} $E[G'] \leftarrow E[G']\cup\{(u,v)\}$
                            $orig[u,v] \leftarrow orig[x,y]$
                            $w[u,v] \leftarrow w[x,y]$
                 {\bf else if} $w[x,y] < w[u,v]$
                 {\bf then} $orig[u,v] \leftarrow orig[x,y]$
                            $w[u,v] \leftarrow w[x,y]$
построить списки смежных вершин $Adj$ для $G'$
{\bf return} $G'$ и $T$
```

a. Пусть T — множество рёбер, возвращённое процедурой MST-Reduce, а T' — минимальный остов графа G', возвращённого этой процедурой. Докажите, что $T \cup \{orig[x,y] : (x,y) \in T'\}$ — минимальный остов графа G.

b. Покажите, что $|V[G']| \leqslant |V|/2$.

c. Покажите, как реализовать процедуру MST-Reduce так, чтобы она исполнялась за время $O(E)$. (Указание. Используйте несложные структуры данных.)

d. Пусть мы подвергли граф k-кратной обработке с помощью процедуры MST-Reduce (выход одного шага является входом следующего). Объясните, почему что общее время выполнения всех k итераций составляет $O(kE)$.

e. Пусть после k применений процедуры MST-Reduce мы воспользовались алгоритмом Прима для сжатого графа. При этом можно выбрать k так, чтобы общее время работы составило $O(E \lg\lg V)$. Почему? Такой выбор асимптотически минимизирует общее время работы. Почему?

f. При каком соотношении $|E|$ и $|V|$ алгоритм Прима с предварительным сжатием эффективнее алгоритма Прима без такого сжатия?

Замечания

Книга Тарьяна [188] содержит обзор задач, связанных с минимальными покрывающими деревьями, и дальнейшую информацию о них. Историю задачи о минимальном покрывающем дереве описали Грэхем и Хелл [92].

Как указывает Тарьян, впервые алгоритм построения минимального остова появился в статье Борувки (O.Borůvka). Алгоритм Крускала опубликован в его статье 1956 года [131]. Алгоритм, известный как алгоритм Прима, действительно изобретён им и описан в [163], но ранее его нашёл Ярник (V.Jarník, 1930).

Возможность использования жадных алгоритмов связана с тем, что рёбра графа образуют матроид, если независимыми считать множества рёбер без циклов (раздел 17.4).

Наиболее быстрым из известных в настоящий момент алгоритмов поиска минимального остова (для случая $|E| = \Omega(V \lg V)$) является алгоритм Прима, реализованный с помощью фибоначчиевой кучи. Для более разреженных графов Фредман и Тарьян [75] приводят алгоритм, время работы которого составляет $O(E\beta(|E|, |V|))$, где $\beta(|E|, |V|) = \min\{i : \lg^{(i)} |V| \leqslant |E|/|V|\}$. Поскольку $|E| \geqslant |V|$, время работы этого алгоритма есть $O(E \lg^* V)$.

25 Кратчайшие пути из одной вершины

Перед нами — карта автомобильных дорог США с обозначенными расстояниями; как выбрать кратчайший маршрут от Чикаго до Бостона?

Можно, конечно, перебрать все возможные маршруты, подсчитать для каждого из них длину и выбрать наименьшую. Но даже если не принимать во внимание маршруты, содержащие циклы, при таком подходе придется перебирать миллионы заведомо негодных вариантов (вряд ли стоит ехать из Бостона в Чикаго через Хьюстон, лежащий на тысячу миль в стороне).

В этой и следующей главах мы рассказываем о том, как можно эффективно решать такие задачи. В *задаче о кратчайшем пути* (shortest-paths problem) нам дан ориентированный взвешенный граф $G = (V, E)$ с вещественной весовой функцией $w\colon E \to \mathbb{R}$. *Весом* (weight) пути $p = \langle v_0, v_1, \ldots, v_k \rangle$ называется сумма весов рёбер, входящих в этот путь:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Вес кратчайшего пути (shortest-path weight) из u в v равен, по определению,

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\}, & \text{если существует путь из } u \text{ в } v; \\ \infty & \text{иначе.} \end{cases}$$

Кратчайший путь (shortest path) из u в v — это любой путь p из u в v, для которого $w(p) = \delta(u, v)$.

В нашем примере с Чикаго и Бостоном можно рассматривать карту дорог как граф, вершинами которого являются перекрёстки, а рёбрами — участки дорог между ними. Вес ребра — это длина участка дороги, и наша задача может состоять в отыскании кратчайшего пути между данным перекрёстком в Чикаго и данным перекрёстком в Бостоне.

Веса не обязаны быть расстояниями: весами могут быть времена, стоимости, штрафы, убытки, … — короче говоря, любая величина, которую мы хотим минимизировать и которая обладает свойством аддитивности.

Алгоритм поиска в ширину на графах (без весовой функции), который мы рассматривали в разд. 23.2, можно рассматривать как решение задачи о кратчайшем пути в частном случае, когда вес каждого ребра равен единице. Многие идеи, связанные с поиском в ширину, будут полезны и для общего случая, поэтому советуем вам ещё раз просмотреть разд. 23.2, прежде чем читать дальше.

Варианты задачи о кратчайшем пути

В этой главе мы рассматриваем только задачу о *кратчайших путях из одной вершины* (single-source shortest-path problem): дан взвешенный граф $G = (V, E)$ и *начальная вершина* v (source vertex); требуется найти кратчайшие пути из s во все вершины $v \in V$. Алгоритм, решающий эту задачу, пригоден и для многих других задач, например:

Кратчайшие пути в одну вершину: дана *конечная* вершина t (destination vertex), требуется найти кратчайшие пути в t из всех вершин $v \in V$. (В самом деле, если обратить все стрелки на рёбрах, эта задача сведется к задаче о кратчайших путях из одной вершины.)

Кратчайший путь между данной парой вершин: даны вершины u и v, найти кратчайший путь из u в v. Разумеется, если мы найдем все кратчайшие пути из u, то тем самым решим и эту задачу; как ни странно, более быстрого способа (который бы использовал тот факт, что нас интересует путь лишь в одну вершину) не найдено.

Кратчайшие пути для всех пар вершин: для каждой пары вершин u и v найти кратчайший путь из u в v. Можно решить эту задачу, находя кратчайшие пути из данной вершины для всех вершин по очереди. (Это, правда, не оптимальный способ — более эффективные подходы мы рассмотрим в следующей главе.)

Рёбра отрицательного веса

В некоторых приложениях веса рёбер могут быть отрицательными. При этом важно, есть ли циклы отрицательного веса. Если из вершины из s можно добраться до цикла отрицательного веса, то потом можно обходить этот цикл сколь угодно долго, и вес будет всё уменьшаться, так что для вершин этого цикла кратчайших путей не существует (рис. 25.1). В таком случае мы будем считать, что вес кратчайшего пути есть $-\infty$.

Если же циклов отрицательного веса нет, то любой цикл можно выбросить, не удлиняя пути. Путей без циклов конечное число, так что вес кратчайшего пути корректно определён.

Во многих задачах нет циклов отрицательного веса не может быть уже потому, что веса всех рёбер неотрицательны. (Таков наш пример с картой автодорог). Некоторые алгоритмы для поиска кратчайших путей (например, алгоритм Дейкстры) исполь-

Рисунок 25.1 25.1 Ориентированный граф с рёбрами отрицательного веса. У каждой вершины указан вес кратчайшего пути из s в эту вершину. Поскольку вершины e и f образуют цикл отрицательного веса, достижимый из вершины s, веса кратчайших путей в каждую из этих вершин равны $-\infty$. Сделав несколько циклов, можно пойти в g, так что вес кратчайшего пути в g также равен $-\infty$. Вершины h, i и j недостижимы из s, так что (хоть они и лежат на цикле отрицательного веса), вес кратчайшего пути в эти вершины есть ∞.

зуют это и применимы лишь для графов с неотрицательными весами. Другие (например, алгоритм Беллмана-Форда) допускают рёбра отрицательного веса и даже дают верный результат, если из исходной вершины нельзя дойти до цикла отрицательного веса. Обычно алгоритм сообщает, если такой цикл отрицательного веса обнаружен.

Представление кратчайших путей в алгоритме

Часто требуется не просто подсчитать веса кратчайших путей, но и найти сами эти пути. Мы будем использовать для их представления тот же приём, что и в деревьях поиска в ширину в разд. 23.2. Именно, пусть $G = (V, E)$ — заданный граф. Для каждой вершины $v \in V$ мы будем помнить её *предшественника* (predecessor) $\pi[v]$. Предшественник вершины — это либо другая вершина (указатель на неё), либо NIL. По завершении работы алгоритмов, рассматриваемых в этой главе, цепочка предшественников, начинающаяся с произвольной вершины v, будет представлять собой кратчайший путь из s в v (в обратном порядке), так что, если $\pi[v] \neq$ NIL, процедура Print-Path(G, s, v) из разд. 23.2 напечатает кратчайший путь из s в v.

В процессе работы алгоритмов цепочки, получаемые итерациями π, не обязательно будут кратчайшими путями, но всё равно можно рассмотреть ориентированный *подграф предшествования* (predecessor subgraph) $G_\pi = (V_\pi, E_\pi)$, определённый так: вершины G_π — это те вершины G, у которых предшественник отличен от NIL, плюс исходная вершина:

$$V_\pi = \{\, v \in V : \pi[v] \neq \text{NIL} \,\} \cup \{s\}.$$

Рёбра G_π — это стрелки, указывающие из $\pi[v] \neq$ NIL в v:

$$E_\pi = \{\, (\pi[v], v) \in E : v \in V_\pi \setminus \{s\} \,\}.$$

Мы докажем, что по окончании работы наших алгоритмов граф G_π будет *"деревом кратчайших путей"* — деревом с корнем s, содержащим кратчайшие пути из s во все достижимые из s вершины. Деревья кратчайших путей аналогичны деревьям поиска в ширину из разд. 23.2, с той разницей, что на сей раз кратчайшими объявляются пути с наименьшим весом, а не наименьшим числом ребер. Точное определение выглядит так. Пусть $G = (V, E)$ — взвешен-

Рисунок 25.2 25.2 (а) Взвешенный ориентированный граф; в вершинах указаны веса кратчайших путей из s. (б) Серые рёбра образуют дерево кратчайших путей с корнем s. (в) Другое дерево кратчайших путей с тем же корнем.

ный ориентированный граф с весовой функцией $w\colon E \to \mathbb{R}$. Предположим, что G не имеет циклов отрицательного веса, достижимых из исходной вершины s, так что все кратчайшие пути из s корректно определены. По определению, *дерево кратчайших путей* (shorted-paths tree) с корнем в s есть ориентированный подграф $G' = (V', E')$, где $V' \subseteq V$ и $E' \subseteq E$, для которого:

1. V' — множество вершин, достижимых из вершины v;
2. G' является деревом с корнем s;
3. для каждого $v \in V'$ путь из s в v в графе G' является кратчайшим путем из s в v в графе G.

Ни кратчайшие пути, ни деревья кратчайших путей не обязаны быть единственными. На рис. 25.2 изображен взвешенный ориентированный граф и два различных дерева кратчайших путей с общим корнем.

План главы

Все алгоритмы для поиска кратчайших путей, рассматриваемые в этой главе, основаны на технике, известной под названием *релаксация* (relaxation). В разд. 25.1 мы рассматриваем общие свойства кратчайших путей, а затем доказываем ряд важных фактов про релаксацию. Алгоритм Дейкстры, решающий задачу о кратчайших путях из одной вершины для случая неотрицательных весов, разобран в разд. 25.2. Разд. 25.3 посвящен алгоритму Беллмана-Форда, применимому и в том случае, когда рёбра могут иметь отрицательный вес. (Если граф содержит цикл отрицательного веса, достижимый из исходной вершины, алгоритм Беллмана-Форда сообщает об этом.) В разд. 25.4 рассматривается алгоритм для нахождения кратчайших путей в ациклических графах, работающий за линейное время. Наконец, в разд. 25.5 мы показываем, как применить алгоритм Беллмана-Форда к решению одного специального случая задачи линейного программирования.

При работе с символами ∞ и $-\infty$ мы будем придерживаться следующих соглашений. Если $a \neq -\infty$, то будем считать, что $a + \infty = \infty + a = \infty$; аналогично, если $a \neq \infty$, то $a + (-\infty) = (-\infty) + a = -\infty$.

25.1 Кратчайшие пути и релаксация

В этом разделе описываются свойства кратчайших путей и объясняется общий приём, используемый всеми алгоритмами этой главы и называемый *"релаксацией"*. Он состоит, грубо говоря, в постепенном уточнении верхней оценки на вес кратчайшего пути в данную вершину — пока неравенство не превратится в равенство.

При первом чтении вы можете разобрать только формулировки результатов (обратите особое внимание на лемму 25.7; леммы 25.8 и 25.9 при первом чтении можно пропустить), после чего перейти к алгоритмам разд. 25.2 и 25.3.

Свойство оптимальности для подзадач

Любая часть кратчайшего пути сама есть кратчайший путь. Это значит, что задача о кратчайших путях обладает свойством оптимальности для подзадач — признак того, что к ней может быть применимо динамическое программирование (глава 16) или жадный алгоритм (глава 17). И действительно, алгоритм Дейкстры является жадным алгоритмом, а алгоритм Флойда-Уоршолла для нахождения кратчайших путей между всеми парами вершин (глава 26) основан на динамическом программировании. Следующая лемма и ее следствие уточняют, в чем конкретно состоит свойство оптимальности для подзадач в задаче о кратчайших путях.

Лемма 25.1 (Отрезки кратчайших путей являются кратчайшими)

Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w\colon E \to \mathbb{R}$. Если $p = \langle v_1, v_2, \ldots, v_k \rangle$ — кратчайший путь из v_1 в v_k и $1 \leqslant i \leqslant j \leqslant k$, то $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ есть кратчайший путь из v_i в v_j.

Доказательство

Если путь p_{ij} не кратчайший, то, заменяя в пути p участок от v_i до v_j на более короткий путь из v_i в v_j, мы уменьшим вес пути из p_1 в p_k — противоречие. (Здесь " *более короткий*" означает "*с меньшим весом*".)

Следствие из доказанной леммы обобщает лемму 23.1.

Следствие 25.2

Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w\colon E \to \mathbf{R}$. Рассммотрим кратчайший путь p из s в v. Пусть $u \to v$ — последнее ребро этого пути (p есть $s \overset{p'}{\leadsto} u \to v$). Тогда $\delta(s, v) = \delta(s, u) + w(u, v)$.

Доказательство

По лемме 25.1 путь p' является кратчайшим, так что $\delta(s, v) = w(p') + w(u, v) = \delta(s, u) + w(u, v)$.

Следующая лемма проста, но полезна.

Лемма 25.3 Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w\colon E \to \mathbb{R}$; пусть $s \in V$. Тогда для всякого ребра $(u, v) \in E$ имеем $\delta(s, v) \leqslant \delta(s, u) + w(u, v)$.

Доказательство.

Вес кратчайшего пути из s в v не превосходит веса любого пути из s в v, в том числе и проходящего на последнем шаге через u.

Релаксация

Техника релаксации, которая уже упоминалась выше, состоит в

Рисунок 25.3 25.3 Релаксация ребра (u, v). В вершинах указаны оценки кратчайшего пути. (а) Поскольку перед релаксацией было $d[v] > d[u] + w(u, v)$, в результате релаксации $d[v]$ уменьшается. (б) Уже до релаксации имеем $d[v] \leqslant d[u] + w(u, v)$. Релаксация ничего не меняет.

следующем. Для каждого ребра $v \in V$ мы храним некоторое число $d[v]$, являющееся верхней оценкой веса кратчайшего пути из s в v; для краткости мы будем называть его просто *оценкой кратчайшего пути* (shortest-path estimate). Начальное значение оценки кратчайшего пути (и предшественников) даётся следующей процедурой:

```
Initialize-Single-Source(G,s)
for (для) всех вершин v \in V[G]
    do d[v] \gets \infty
       \pi[v] \gets \text{\sc nil}
d[s] \gets 0
```

Иными словами, первоначально $\pi[v] = $ NIL для всех v; при этом $d[s] = 0$ и $d[v] = \infty$ для остальных вершин v.

Релаксация ребра $(u, v) \in E$ состоит в следующем: значение $d[v]$ уменьшается до $d[u] + w(u, v)$ (если второе значение меньше первого): при этом $d[v]$ остаётся верхней оценкой в силу леммы 25.3. Мы хотим, чтобы $\pi[v]$ указывали на путь, использованный при получении этой верхней оценки, поэтому одновременно мы меняем значение $\pi[v]$:

```
Relax(u,v,w)
if d[v] > d[u] + w(u,v)
   then d[v] \gets d[u] + w(u,v)
        \pi[v] \gets u
```

На рис. 25.3 приведены два примера релаксации: в одном случае оценка кратчайшего пути уменьшается, в другом ничего не происходит.

Алгоритмы, описываемые в этой главе, устроены так: они вызывают процедуру INITIALIZE-SINGLE-SOURCE, а затем производят релаксацию рёбер. Разные алгоритмы отличаются порядком, в котором рёбра подвергаются релаксации. В алгоритме Дейкстры и алгоритме для ациклических графов каждое ребро подвергается релаксации лишь единожды. В алгоритме Беллмана-Форда рёбра подвергаются релаксации по нескольку раз.

Свойства релаксации

Леммы, доказываемые в этом и следующем разделах, будут использованы при доказательстве правильности алгоритмов поиска кратчайших путей.

Следующая лемма очевидна.

Лемма 25.4

Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w\colon E \to \mathbb{R}$, и пусть $(u, v) \in E$. Тогда сразу же после релаксации этого ребра (вызова Relax(u, v, w)) выполняется неравенство $d[v] \leqslant d[u] + w(u, v)$.

Лемма 25.5

Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией w; пусть $s \in V$ — начальная вершина. Тогда после выполнения процедуры Initialize-Single-Source(G, s), а затем произвольной последовательности операций релаксации рёбер, для каждой вершины $v \in V$ выполнено неравенство $d[v] \geqslant \delta(s, v)$. Если при этом для какой-то из вершин v это неравенство обращается в равенство, то равенство $d[v] = \delta(s, v)$ останется верным и в дальнейшем (при последующих релаксациях рёбер).

Доказательство

В самом деле, после инициализации значения $d[v]$ бесконечны при $v \neq s$ (и потому являются оценкой сверху для чего угодно), а $d[s] = 0$ (что тоже правильно). В процессе релаксации значение $d[v]$ остаётся верхней оценкой для $\delta(s, v)$, поскольку

$$\delta(s, v) \leqslant \delta(s, u) + w(u, v) \leqslant d[u] + w(u, v)$$

(первое неравенство — по лемме 25.3).

Второе утверждение леммы: поскольку в процессе релаксации значения d могут только уменьшаются, а после достижения равенства $d[v] = \delta(s, v)$ дальше уменьшаться некуда.

Следствие 25.6

Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией w и исходной вершиной s. Пусть вершина $v \in V$ недостижима из s. Тогда после исполнения процедуры Initialize-Single-Source(G, s) и любой последовательности релаксаций ребер значение $d[v]$ будет оставаться бесконечным (и равным $\delta(s, v)$).

Следующая лемма играет основную роль при доказательстве правильности алгоритмов поиска кратчайших путей.

Лемма 25.7

Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией w и исходной вершиной s. Пусть $s \rightsquigarrow u \to v$ — кратчайший путь с последним ребром (u, v) Предположим, что была исполнена процедура Initialize-Single-Source(G, s), а затем — последовательность релаксаций некоторых ребер, включающая релаксацию ребра (u, v). Если в какой-то момент до релаксации ребра (u, v) выполнялось равенство $d[u] = \delta(s, u)$, то в любой момент после релаксации (u, v) будет выполнено равенство $d[v] = \delta(s, v)$.

Доказательство

В самом деле, равенство $d[u] = \delta(s, u)$ сохранится до момента релаксации ребра (u, v) (как, впрочем, и до любого дальнейшего момента, см. лемму 25.5). Поэтому сразу после релаксации ребра

(u, v) имеем

$$d[v] \leqslant d[u] + w(u, v) = \delta(s, u) + w(u, v) = \delta(s, v)$$

(последнее равенство — по следствию 25.2). Поскольку, с другой стороны, $d[v] \geqslant \delta(s, v)$ по лемме 25.5, получаем, что $d[v] = \delta(s, v)$ сразу после релаксации (а значит, и позже).

Деревья кратчайших путей

Посмотрим, что происходит при наших операциях с подграфом предшествования G_π.

Лемма 25.8

Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией w и исходной вершиной s, причем в графе G нет циклов отрицательного веса, достижимых из s. Тогда после операции Initialize-Single-Source(G, s), за которой следует произвольная последовательность релаксаций рёбер, подграф предшественников G_π является деревом с корнем s.

Доказательство

Напомним, что вершинами графа G_π являются те вершины $v \in V$, для которых $\pi[v] \neq$ NIL, а также вершина s. Другими словами, в него входят те вершины v, для которых $d[v]$ конечно (чтобы убедиться в этом, достаточно посмотреть на процедуру релаксации: при уменьшении $d[v]$ происходит присваивание переменной $\pi[v]$). Для каждой вершины v графа G_π в этот граф включается ребро с началом $\pi[v]$ и концом v. По построению это ребро является ребром исходного графа G.

Сразу после инициализации граф G_π состоит только из начальной вершины и тем самым является деревом. Мы будем доказывать по индукции, что он остаётся деревом после выполнения произвольной последовательности операций релаксации.

Когда в нём появляются новые вершины? Это происходит при выполнении операции релаксации ребра (u, v), до которой $d[v]$ было бесконечным (а стало конечным — после релаксации любого ребра (x, y) значение $d[y]$ обязательно конечно). В этот момент $\pi[v]$ становится равным u, то есть к дереву G_π добавляется лист. При этом оно остаётся деревом.

Остаётся проверить, что граф G_π остаётся деревом и в том случае, когда при релаксации ребра (u, v) значение $d[v]$ уменьшается от одного конечного значения до другого. Давайте посмотрим, что происходит с G_π при релаксации такого ребра (u, v). Поддерево с корнем в v отрезается (от прежнего родителя вершины v) и прививается к вершине u ($\pi[v]$ становится равным u). Это может нарушить структуру дерева только в том случае, если вершина u была потомком вершины v, то есть лежала в поддереве с корнем v (при этом образуется цикл, см. рис. 25.4)

Нам остаётся убедиться, что такого произойти не может. Проверим, что в этом случае образующийся цикл (от v к u в дереве

Рис.25.4 НОВЫЙ: эскиз см. на полях книги

Рисунок 25.4 25.4. Плохой случай: релаксация ребра (u, v), где вершина u является потомком v в дереве предшествования. Если такое проивходит, возникает цикл, имеющий отрицательную сумму весов.

предшествования, а затем по ребру (u, v)) имеет отрицательную сумму весов.

Поскольку ребро (u, v) подверглось релаксации, до её выполнения имело место неравенство $d[u] + w(u, v) < d[v]$, или $(d[u] - d[v]) + w(u, v) < 0$. Мы сейчас покажем, что вес пути от v в u в графе G_π не превосходит $d[u] - d[v]$, и тем самым найдём цикл отрицательного веса в графе G, достижимый из s, которого по предположению не существует.

Итак, почему же вес пути в G_π от вершины v к вершине u не превосходит разницы значений функции d в конце и начале пути? Очевидно, достаточно проверить это для одного ребра, то есть проверить, что если ребро (p, q) в какой-то момент входит в G_π, то в этот момент

$$w(p, q) \leqslant d[q] - d[p] \tag{25.1}$$

то есть $d[q] \geqslant d[p] + w(p, q)$. Непосредственно после релаксации ребра (p, q) это неравенство обращается в равенство. Затем величины $d[p]$ и $d[q]$ могут уменьшаться. Если уменьшается $d[p]$, то неравенство не нарушается. Если уменьшается $d[q]$, это значит, что происходит релаксация ребра, ведущего в q, при этом меняется и $\pi[q]$ и для нового ребра $(\pi[q], q)$ выполнено неравенство (25.1). Это рассуждение завершает доказательство леммы 25.8

Пусть в результате последовательности релаксаций мы добились того, что $d[v] = \delta(s, v)$ для всех вершин v. Покажем, что в этом случае граф G_π является деревом кратчайших путей.

Лемма 25.9

Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией w и исходной вершиной s, причем в графе G нет циклов отрицательного веса, достижимых из s. Предположим, что после операции INITIALIZE-SINGLE-SOURCE(G, s), за которой следует некоторая последовательность релаксаций ребер, оказалось, что $d[v] = \delta(s, v)$ для всех $v \in V$. Тогда подграф предшествования G_π является деревом кратчайших путей.

Доказательство

Согласно определению, нам надо проверить, что $V[G_\pi]$ совпадает со множеством вершин графа G, достижимых из s, что G_π — дерево с корнем s, и что пути в G_π из s в его вершины являются кратчайшими путями в G.

Второе из этих утверждений есть лемма 25.8; из неё же следует, что все вершины G_π достижимы из s (даже в подграфе G_π); обратно, если вершина $v \neq s$ достижима в графе G из s, то

$d[v] = \delta(s, v) < \infty$, так что имела место релаксация ребра с концом v и $\pi[v] \neq$ NIL, то есть $v \in V[G_\pi]$. Этим доказано первое утверждение.

Докажем третье утверждение. Неравенство (25.1) гарантирует, что длина пути от s до v в дереве G_π не превосходит $d[v] - d[s] = d[v] = \delta(s, v)$, так что пусть этот — кратчайший.

Упражнения

25.1-1 Укажите ещё два дерева кратчайших путей для графа рис. 25.2.

25-1.2 Приведите пример взвешенного ориентированного графа $G = (V, E)$ с исходной вершиной s, обладающего следующим свойством: для каждого ребра $(u, v) \in E$ существует как дерево кратчайших путей с корнем s, содержащее ребро (u, v), так и дерево кратчайших путей с корнем s, указанного ребра не содержащее.

25-1.3 Убедитесь, что доказательство леммы 25.3 проходит и для того случая, когда веса путей равны ∞ или $-\infty$.

25-1.4 Пусть $G = (V, E)$ — взвешенный ориентированный граф с исходной вершиной s. Предположим, что после операции INITIALIZE-SINGLE-SOURCE(G, s), за которой следует некоторая последовательность релаксаций ребер, оказалось, что $\pi[s] \neq$ NIL. Докажите, что G содержит цикл отрицательного веса.

25-1.5 Пусть $G = (V, E)$ — взвешенный ориентированный граф, в котором веса всех ребер неотрицательны. Выберем исходную вершину $s \in V$, а для каждой $v \in V \setminus \{s\}$ выберем вершину $\pi[v] \in V$ таким образом, чтобы $\pi[v]$ была предшественником v на некотором кратчайшем пути из s в v; если v недостижима из s, положим $\pi[v] =$ NIL. Приведите пример графа G и функции π, удовлетворяющих приведенным выше условиям, для которых подграф G_π, построенный по функции π, содержит циклы (в силу леммы 25.8, инициализация с последовательностью релаксаций такой функции π дать не может).

25-1.6 Пусть $G = (V, E)$ — взвешенный ориентированный граф с начальной вершиной s, не имеющий циклов отрицательного веса, достижимых из s. Покажите, что после операции INITIALIZE-SINGLE-SOURCE(G, s), за которой следует произвольная последовательность релаксаций рёбер, всякая вершина $v \in V[G_\pi]$ достижима из s в графе G_π.

25-1.7 Пусть $G = (V, E)$ — взвешенный ориентированный граф с начальной вершиной s, не содержащий циклов отрицательного веса. Покажите, что можно провести операцию INITIALIZE-SINGLE-SOURCE(G, s), а затем $|V| - 1$ релаксаций ребер таким образом, что в результате равенство $d[v] = \delta(s, v)$ будет выполняться для всех $v \in V$.

25-1.8 Пусть $G = (V, E)$ — взвешенный ориентированный граф с начальной вершиной s. Предположим, что из вершины s достижим цикл отрицательного веса. Покажите, что существует бесконечная последовательность релаксаций, после каждой из которых функция

Рис. 25.5

Рисунок 25.5 Рис. 25.5. Алгоритм Дейкстры. Исходная вершина — крайняя левая. Оценки кратчайшего пути указаны в вершинах. Серым цветом выделены рёбра (u, v), для которых $\pi[v] = u$. Чёрные вершины лежат в множестве S, остальные находятся в очереди $Q = V \setminus S$. (а) Перед первой итерацией цикла **while**. Серая вершина имеет минимальное значение d и выбирается в качестве вершины u в строке 5 (б–е) Последовательные состояния после каждой итерации цикла **while**. Серая вершина выбирается в качестве вершины u при следующей итерации. Значения d и π на рисунке (е) — окончательные.

d меняется.

Алгоритм Дейкстры решает задачу о кратчайших путях из одной вершины для взвешенного ориентированного графа $G = (V, E)$ с исходной вершиной s, в котором веса всех ребер неотрицательны. ($w(u, v) \geqslant 0$ для всех $(u, v) \in E$). В этом разделе мы рассматриваем только такие графы.

В процессе работы алгоритма Дейкстры поддерживается множество $S \subseteq V$, состоящее из вершин v, для которых $\delta(s, v)$ уже найдено (то есть $d[v] = \delta(s, v)$). Алгоритм выбирает вершину $u \in V \setminus S$ с наименьшим $d[u]$, добавляет u к множеству S и производит релаксацию всех рёбер, выходящих из u, после чего цикл повторяется. Вершины, не лежащие в S, хранятся в очереди с приоритетами Q, определяемыми значениями функции d. Предполагается, что граф задан с помощью списков смежных вершин.

```
Dijkstra(G,w,s)
1 Initialize-Single-Source(G,s)
2 S \gets \emptyset
3 Q \gets V[G]
4 while Q \ne \emptyset
5       do u \gets Extract-Min(Q)
6          S \gets S \cup \{u\}
7          for (для) всех вершин v\in Adj[u]
8              do Relax(u,v,w)
```

Работа алгоритма Дейкстры показана на рис. 25.5. В строке 1 инициализируются d и π, а в строках 2 и 3 инициализируются множество S (как пустое) и очередь $Q = V \setminus S = V$. При каждом исполнении цикла в строках 4–8 из очереди $Q = V \setminus S$ изымается вершина u с наименьшим значением $d[u]$; она добавляется к множеству S (в первый раз имеем $u = s$). В строках 7–8 каждое ребро (u, v), выходящее из u, подвергается релаксации (при этом могут измениться оценка $d[v]$ и предшественник $\pi[v]$). Заметим, что в цикле новые вершины в очередь Q не добавляются и что каждая вершина, удаляемая из Q, добавляется к множеству S лишь однажды. Следовательно, число итераций цикла **while** равно $|V|$.

Поскольку алгоритм Дейкстры всякий раз выбирает для обработки вершины с наименьшей оценкой кратчайшего пути, можно в этом случае картинка осталась той же, но подпись переписана

Рисунок 25.6 25.6 В пути от s до u выделена часть (s–x), проходящая целиком внутри S, и первая вершина y, лежащая вне S (возможно, $x = s$ или $y = u$).

сказать, что он относится к жадным алгоритмам (гл. 17). Покажем, что в данном случае жадная стратегия даёт правильный результат.

Теорема 25.10 (Правильность алгоритма Дейкстры)

Пусть $G = (V, E)$ — взвешенный ориентированный граф с неотрицательной весовой функцией $w\colon E \to \mathbb{R}$ и исходной вершиной s. Тогда после применения алгоритма Дейкстры к этому графу для всех вершин $u \in V$ будут выполняться равенства $d[u] = \delta(s, u)$.

Доказательство

Проверим, что после любого числа итераций цикла textbfwhile выполнено следующее свойство:

(a) для вершин $v \in S$ значение $d[v]$ равно $\delta(s, v)$, причём существует путь из s в v веса $\delta(s, v)$, проходящий только по вершинам из S;

(b) для вершин $v \in Q = V \setminus S$ значение $d[v]$ равно наименьшему весу пути из s в v, если учитывать только те пути, в которых все вершины, кроме последней (v), лежат в S (если таких путей нет, $d[v] = \infty$).

После первой итерации цикла (когда в S лежит только вершина s и только что проведены релаксации всех рёбер, ведущих из s) это свойство выполнено. Проверим, что оно не нарушится и на следующих итерациях.

Пусть u — вершина Q с минимальным значением $d[u]$. Если $d[u] = \infty$, то не существует пути, в котором все вершины, кроме последней, лежат в S. Значит, из S вообще нельзя никак выйти (оборвав путь в момент выхода, мы получили бы путь с указанным свойством). Пусть теперь $d[u]$ конечно. Тогда существует путь из s в u веса $d[u]$, проходящий по вершинам S до последнего момента (до вершины u). Докажем, что он будет кратчайшим. (Тем самым вершину u можно добавить в S, не нарушая утверждения (a).)

Пусть есть какой-то другой путь из s в u (рис. 25.6). Посмотрим на первую вершину этого пути, лежащую вне S; обозначим её y (возможно, $y = u$). Часть пути от s до y проходит по S до последнего момента, поэтому вес этой части не меньше $d[y]$ (по предположению (a)). Осталось заметить, что вес всего пути не меньше веса его части (веса́ рёбер неотрицательны) и что $d[y] \leqslant d[u]$, поскольку в Q вершина u имела наименьшее значение $d[u]$.

Итак, условие (a) останется верным после добавления u к S. Остаётся проверить условие (b). Поскольку S увеличилось и стало равным $S' = S \cup \{u\}$, путей, не выходящих из S до последнего момента, стало больше, и для соблюдения условия (b) значения $d[v]$ для $v \in Q$ надо уменьшать. Это и делается в процессе релаксации.

При релаксации ребра (u, v) значение $d[v]$ становится равным

$$d'[v] = \min(d[v], d[u] + w(u, v)) \tag{25.2}$$

Мы знаем, что $d[v]$ есть вес некоторого пути, который ведёт из s в v, оставаясь до последнего шага в S. Второй член $d[u] + w(u, v)$ есть вес некоторого пути, который ведёт из s в u (оставаясь до последнего шага в S), а затем идёт по ребру (u, v). Таким образом существует путь из s в v, который до последнего шага остаётся в S' и имеет вес $d'[v]$. Покажем, что любой путь p из s в v, который до последнего шага остаётся в S', имеет вес не меньше $d'[v]$. В самом деле, если $x \in S'$ — его предпоследняя вершина, то либо $x \in S$, и тогда вес пути p не меньше $d[v]$ по индуктивному предположению (a), либо $x = u$, и тогда вес пути p не меньше $d[u] + w(u, v)$.

Мы доказали, что условия (a) и (b) остаются верными после любого числа итераций. Значит, они верны и после выхода из цикла; в этот момент $S = V$ и потому $d[u] = \delta(s, u)$ для любой вершины $v \in V$.

Из доказанной теоремы и леммы 25.9 немедленно вытекает

Следствие 25.11

Пусть $G = (V, E)$ — взвешенный ориентированный граф с неотрицательной весовой функцией w и исходной вершиной s. Тогда после применения алгоритма Дейкстры к этому графу подграф предшественников G_π будет деревом кратчайших путей с корнем в s.

Время работы алгоритма Дейкстры

Сначала предположим, что очередь с приоритетами $Q = V \setminus S$ реализована как массив. При этом стоимость операции EXTRACT-MIN есть $O(V)$; поскольку алгоритм делает V таких операций, суммарная стоимость всех удалений из очереди есть $O(V^2)$. Что же до стоимости остальных операций, то каждая вершина $v \in V$ добавляется к множеству S только один раз, и каждое ребро из $Adj[v]$ обрабатывается тоже один раз. Общее количество элементов во всех списках смежных вершин есть $|E|$; стоимость каждой релаксации есть $O(1)$. Поэтому стоимость прочих операций есть $O(E)$, а общая стоимость алгоритма есть $O(V^2 + E)$.

Если граф является разреженным, имеет смысл реализовать очередь с помощью (двоичной) кучи (раздел 7.5). Получаемый алгоритм называют иногда *модифицированным алгоритмом Дейкстры* (modified Dijkstra algorithm). Стоимость операции EXTRACT-MIN при такой реализации очереди есть $O(\lg V)$, а стоимость построения кучи (строка 3) есть $O(V)$. Присваивание $d[v] \leftarrow d[u] + w(u, v)$ при релаксации ребра реализуется с помощью вызова процедуры DECREASE-KEY$(Q, v, d[u] + w(u, v))$, имеющего стоимость $O(\lg V)$ (см. упражнение 7.5-4). Количество таких вызовов не превосходит $|E|$, так что общая стоимость модифицированного алгоритма Дейкстры есть $O((V + E) \lg V)$ (если все вершины достижимы из исходной, то граф связен, $|E| \geqslant |V| - 1$ и последнюю оценку можно переписать как $O(E \lg V))$.

Если реализовать очередь Q в виде фибоначчиевой кучи (см. главу 21), то можно добиться стоимости $O(V \lg V + E)$: в самом деле, при этом учётная стоимость каждой из $|V|$ операций EXTRACT-MIN будет $O(\lg V)$, а учетная стоимость каждой из $|E|$ операций DECREASE-KEY будет $O(1)$. Кстати, кучи Фибоначчи были изобретены именно в связи с модифицированным алгоритмом Дейкстры: поскольку при его исполнении может потребоваться гораздо больше вызовов процедуры DECREASE-KEY, чем EXTRACT-MIN, хотелось снизить стоимость DECREASE-KEY.

Алгоритм Дейкстры напоминает как алгоритм поиска в ширину (разд. 23.2), так и алгоритм Прима нахождения минимального покрывающего дерева (разд. 24.2). Роль множества S в алгоритме Дейкстры аналогична роли множества черных вершин при поиске в ширину. Сходство алгоритмов Дейкстры и Прима в том, что они оба пользуются очередью с приоритетами при реализации жадной стратегии.

Упражнения

25.2-1

Примените алгоритм Дейкстры к графу на рис. 25.2, выбрав за исходную сначала вершину s, а затем вершину y. Следуя образцу рис. 25.5, покажите ход исполнения алгоритма.

25-2.2

Приведите пример графа (с отрицательными весами рёбер), для которого алгоритм Дейкстры даёт неверный ответ. Где в доказательстве теоремы 25.10 используется неотрицательность весовой функции?

25-2.3

Если заменить строку 4 в алгоритме Дейкстры на

```
4 while |Q| >1
```

то число итераций цикла **while** уменьшится на 1. Будет ли изменённый таким образом алгоритм правильным?

25-2.4

Пусть с каждым ребром ориентированного графа $G = (V, E)$ ассоциировано действительное число $r(u, v)$, причём $0 \leqslant r(u, v) \leqslant 1$. Будем интерпретировать $r(u, v)$ как *"надёжность"* — вероятность того, что сигнал успешно пройдет по каналу из u в v. Считая, что события прохождения сигнала по различным каналам независимы, предложите алгоритм для нахождения наиболее надёжного пути между двумя данными вершинами.

25-2.5

Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w \colon E \to \{0, 1, \ldots, W - 1\}$, где $W > 0$ — целое число. Модифицируйте для данного случая алгоритм Дейкстры так, чтобы он искал кратчайшие пути из данной вершины за время $O(WV+E)$.

25-2.6

Усовершенствуйте решение предыдущего упражнения, сделав время работы алгоритма равным $O((V+E)\lg W)$. (Указание: сколько различных оценок кратчайшего пути для вершин из $V \setminus S$ может встретиться одновременно?)

25.2 Алгоритм Беллмана-Форда

Алгоритм Беллмана-Форда (Bellman-Ford algorithm) решает задачу о кратчайших путях из одной вершины для случая, когда весам ребер разрешено быть отрицательными. Этот алгоритм возвращает значение TRUE, если в графе нет цикла отрицательного веса, достижимого из исходной вершины, и FALSE, если таковой цикл имеется. В первом случае алгоритм находит кратчайшие пути и их веса; во втором случае задача кратчайших путей (по крайней мере для некоторых вершин) не существует.

Подобно алгоритму Дейкстры, алгоритм Беллмана-Форда производит релаксацию рёбер, пока все значения $d[v]$ не сравняются с $\delta(s,v)$ (если все $\delta(s,v)$ определены).

```
Bellman-Ford(G,w,s)
1 Initialize-Single-Source(G,s)
2 for i \gets 1 to |V[G]|-1
3     do for (для) каждого ребра (u,v) \in E[G]
4            do Relax(u,v,w)
5 for (для) каждого ребра (u,v) \in E[G]
6     do if d[v]>d[u]+w(u,v)
7        then return \textsc{false}
8 return \textsc{true}
```

На рис. 25.7 показана работа алгоритма Беллмана-Форда для графа с пятью вершинами. После инициализации (строка 1) алгоритм $|V|-1$ раз делает одно и то же: перебирает по разу все рёбра графа и подвергает каждое из них релаксации (строки 2–4). После этого алгоритм проверяет, нет ли цикла отрицательного веса, достижимого из начальной вершины s (строки 5–8; вскоре мы увидим, почему эта проверка позволяет выявить такой цикл).

Время работы алгоритма Беллмана-Форда есть $O(VE)$, поскольку общее число релаксаций есть $O(VE)$, стоимость инициализации в строке 1 есть $O(V)$, а стоимость цикла в строках 5–8 есть $O(E)$.

Докажем, что алгоритм Беллмана-Форда работает правильно.

Лемма 25.12

Пусть $G=(V,E)$ — взвешенный ориентированный граф с весо-

Рисунок 25.7 25.7 Алгоритм Беллмана-Форда. Исходная вершина — крайняя левая (z). Оценки кратчайшего пути указаны в вершинах. Серым цветом выделены рёбра (u, v), для которых $\pi[v] = u$. В данном примере при каждой итерации цикла в строках 2–4 рёбра подвергаются релаксации в лексикографическом порядке: (u, v), (u, x), (u, y), (v, u), (x, v), (x, y), (y, v), (y, z), (z, u), (z, x). (а) Перед первым обходом рёбер. (б–д) Последовательные состояния после каждой обработки всех рёбер. Значения d на рисунке (д) — окончательные. В данном случае алгоритм Беллмана-Форда возвращает значение TRUE.

вой функцией w и исходной вершиной s, не содержащий циклов отрицательного веса, достижимых из s. Тогда по окончании работы процедуры Bellman-Ford равенство $d[v] = \delta(s, v)$ будет выполняться для всех вершин v, достижимых из s.

Доказательство Пусть $p = \langle s = v_0, v_1, \ldots, v_k = v\rangle$ — кратчайший путь из s в v. Можно считать, что этот путь не содержит циклов (они только увеличивают вес по предположению), поэтому $k \leqslant |V| - 1$. Докажем индукцией по i, что после i-ой итерации цикла (в строках 2–4) будет выполнено равенство $d[v_i] = \delta(s, v_i)$: поскольку всего делается $|V| - 1$ итераций, лемма будет следовать из этого утверждения и леммы 25.5.

При $i = 0$ это очевидно, так как $d[s] = \delta(s, s) = 0$ при входе в цикл. Пусть после $i - 1$-ой итерации было $d[v_{i-1}] = \delta(s, v_{i-1})$. При i-ой итерации произойдет релаксация ребра (v_{i-1}, v_i), после чего по лемме 25.7 установится равенство $d[v_i] = \delta(s, v_i)$. (Другие релаксации могут также уменьшать $d[v_i]$, но не могут сделать его меньше $\delta(s, v_i)$.)

Следствие 25.13

Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией w и исходной вершиной s. Тогда вершина $v \in V$ достижима из s в том и только том случае, когда по окончании работы процедуры Bellman-Ford, применённой к этому графу, оказывается, что $d[v] < \infty$.

Доказательство оставляется читателю (упр. 25.3-2).

Теорема 25.14 (Правильность алгоритма Беллмана-Форда)

Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией w и исходной вершиной s. Если процедура Bellman-Ford, примененная к этому графу, возвращает значение TRUE, то в результате её работы для всех $v \in V$ будут выполнены равенства $d[v] = \delta(s, v)$ и подграф предшественников G_π будет деревом кратчайших путей с корнем s. Если же процедура Bellman-Ford возвращает значение FALSE, то в графе есть цикл отрицательного веса, достижимый из вершины s.

Доказательство Если цикла отрицательного веса, достижимого из вершины s, в графе нет, то в результате работы процедуры Bellman-Ford будем иметь $d[v] = \delta(s, v)$ для всех $v \in V$ (лемма 25.12 и следствие 25.13); следовательно, граф G_π будет деревом кратчайших путей с вершиной s (лемма 25.9). Коль скоро $d[v] = \delta(s, v)$ для всех $v \in V$, из леммы 25.3 следует, что для всякого ребра (u, v) выполнено неравенство $d[v] \leqslant d[u] + w(u, v)$. Значит, ни одно из условий в строке 6 алгоритма выполнено не будет, и алгоритм возвратит значение TRUE.

Пусть теперь в графе есть цикл отрицательного веса $\langle v_0, v_1, \dots, v_k = v_0 \rangle$, достижимый из исходной вершины; нам надо показать, что алгоритм возвратит значение FALSE. В самом деле, если $d[v_i] \leqslant d[v_{i-1}] + w(v_{i-1}, v_i)$ для всех $i = 1, 2, \dots, k$, то, складывая эти k неравенств и сокращая $\sum d[v_i]$ в обеих частях, получим $0 \leqslant \sum_{i=1}^{k} w(v_{i-1}, v_i)$, что противоречит выбору цикла. Значит, для некоторого i имеем $d[v_i] > d[v_{i-1}] + w(v_{i-1}, v_i)$, и алгоритм возвращает значение FALSE.

Упражнения

25.3-1 Примените алгоритм Беллмана-Форда к ориентированному графу рис. 25.7, выбрав y в качестве исходной вершины. Рёбра обрабатывайте в лексикографическом порядке; изобразите этапы выполнения алгоритма как на рис. 25.7. Замените вес ребра (y, v) на 4 и проделайте то же самое, выбрав исходной вершиной z.

25.3-2 Докажите следствие 25.13.

25-3.3 Пусть во взвешенном ориентированном графе нет циклов отрицательного веса. Определим число m следующим образом: для каждой пары вершин $u, v \in V$, для которой существует путь из u в v, найдем минимальное количество ребер в путях минимального веса, идущих из u в v; затём возьмем максимум этих чисел по всем таким парам — это и есть m. Покажите, что в алгоритме Беллмана-Форда достаточно выполнять цикл в строках 2–4 m раз.

25.3-4 Модифицируйте алгоритм Беллмана-Форда таким образом, чтобы и для вершин v, у которых $\delta(s, v) = -\infty$ (как мы помним, такое бывает тогда и только тогда, когда существует путь из s в v, задевающий цикл отрицательного веса), после исполнения алгоритма было установлено правильное значение $d[v] = \delta(s, v) = -\infty$.

25.3-5 Пусть $G = (V, E)$ — взвешенный ориентированный граф. Разработайте алгоритм, работающий за время $O(VE)$ и находящий для каждой вершины $v \in V$ значение $\delta^*(v) = \min_{u \in V}\{\delta(u, v)\}$. (В графе могут быть рёбра с отрицательным весом.)

25-3.6 Пусть взвешенный ориентированный граф $G = (V, E)$ имеет цикл отрицательного веса. Разработайте алгоритм, печатающий вершины такого цикла (хотя бы одного).

Рис. 25.8 (в оригинале в подписи была опечатка: v, тогда как надо u).

Рисунок 25.8 25.8. Алгоритм для поиска кратчайших путей в ациклическом ориентированном графе. Вершины топологически отсортированы (на рисунке слева направо). Исходная вершина — s. В вершинах записаны значения функции d; для серых рёбер (u, v) имеем $\pi[v] = u$. (а) Перед первой итерацией цикла в строках 3–5. (б–ж) Последовательные состояния после каждой итерации цикла в строках 3–5. На каждом из этих рисунков прибавляется по одной чёрной вершине (которая была вершиной u во время соответствующей итерации цикла). Значения d на рисунке (ж) — окончательные.

25.3 Кратчайшие пути в ациклическом ориентированном графе

В ациклическом ориентированном графе $G = (V, E)$ кратчайшие пути из одной вершины можно найти за время $O(V + E)$, если проводить релаксацию ребер в порядке, заданном топологическим упорядочением вершин. Заметим, что в ациклическом ориентированном графе кратчайшие пути всегда определены, поскольку циклов отрицательного веса (и вообще циклов) нет.

В разделе 23.4 мы рассматривали алгоритм топологической сортировки вершин ациклического графа. Он располагает их в таком порядке, чтобы все рёбра графа вели от ”*меньших*” вершин к ”*большим*” (в смысле этого порядка). После этого мы просматриваем вершины в этом порядке и для каждой вершины подвергаем релаксации все выходящие из неё рёбра.

```
Dag-Shortest-Paths(G,w,s)
1 топологически отсортировать вершины G
2 Initialize-Single-Source(G,s)
3 for (для) всех вершин u (в найденном порядке)
4     do for (для) всех вершин v \in Adj[u]
5            do Relax(u,v,w)
```

Пример работы этого алгоритма приведён на рис. 25.8.

Оценим время работы алгоритма Dag-Shortest-Paths. Как мы видели в разд. 23.4, стоимость топологической сортировки (строка 1) есть $\Theta(V + E)$, а стоимость инициализации в строке 2 есть $O(V)$. В цикле в строках 3–5 каждое ребро обрабатывается, как и в алгоритме Дейкстры, ровно один раз, но, в отличие от алгоритма Дейкстры, стоимость такой обработки есть $O(1)$. Стало быть, наш алгоритм выполняется за время $\Theta(V + E)$, пропорциональное объему памяти, необходимому на представление графа с помощью списков смежных вершин.

Покажем, что алгоритм Dag-Shortest-Paths правилен.

Теорема 25.15

Пусть взвешенный ориентированный граф $G = (V, E)$ с исходной вершиной s не содержит циклов. Тогда по окончании работы процедуры DAG-SHORTEST-PATHS для всех $v \in V$ будут выполнены равенства $d[v] = \delta(s, v)$, а подграф предшественников G_π будет деревом кратчайших путей.

Доказательство

Согласно лемме 25.9, достаточно доказать равенства $d[v] = \delta(s, v)$. Если вершина v недостижима из s, то $d[v] = \delta(s, v) = \infty$ по следствию 25.6. Пусть теперь вершина v достижима из s и $p = \langle s = v_0, v_1, \ldots, v_k = v \rangle$ — кратчайший путь. После топологической сортировки вершины этого пути расположены как раз в указанном порядке, так что ребро, (v_0, v_1) подвергалось релаксации до ребра (v_1, v_2), которое предшествовало ребру (v_2, v_3) и т.д. Индукция по i с использованием леммы 25.7 (как в доказательстве корректности алгоритма Беллмана-Форда) показывает теперь, что $d[v_i] = \delta(s, v_i)$ для всех i, что и требовалось доказать.

Интересное приложение описанного алгоритма — нахождение критических путей в смысле так называемой *"технологии PERT"* (program evaluation and review technique). В этом приложении каждое ребро ациклического ориентированного графа обозначает какое-то дело, а вес ребра есть время, необходимое на его выполнение. Если имеются рёбра (u, v) и (v, x), то работа, соответствующая ребру (u, v), должна быть выполнена до начала работы, соответствующей ребру (v, x). *Критический путь* (critical path) — это *длиннейший* путь в графе; его вес равен времени, которое будет затрачено на выполнение всех работ, если мы по максимуму используем возможность выполнять некоторые работы параллельно. Чтобы найти критический путь, можно поменять знак у всех весов на противоположный и запустить алгоритм DAG-SHORTEST-PATHS.

Упражнения

25-4.1 Примените алгоритм DAG-SHORTEST-PATHS к графу рис. 25.8, выбрав вершину r в качестве исходной.

25-4.2 Докажите, что алгоритм DAG-SHORTEST-PATHS останется правильным, если обрабатывать только первые $|V| - 1$ вершин.

25-4.3 В задаче о планировании работ по технологии PERT можно рисовать граф иначе, считая, что работы соответствуют не рёбрам, а вершинам графа. При этом каждой вершине присвоен вес (требуемое время), а стрелки указывают определяли последовательность работ (ребро (u, v) требует завершить работу u до начала работы v). Модифицируйте алгоритм DAG-SHORTEST-PATHS так, чтобы он за линейное время находил в ациклическом ориентированном графе путь с максимальной суммой весов вершин.

25-4.4 Разработайте эффективный алгоритм, подсчитывающий общее число путей в ациклическом ориентированном графе, и оцените время его работы.

25.4 Ограничения на разности и кратчайшие пути

Общая задача линейного программирования состоит в отыскании экстремума линейной функции на множестве, заданном системой линейных неравенств. В этом разделе мы рассмотрим специальный случай задачи линейного программирования, сводящийся к нахождению кратчайших путей из одной вершины. Таким образом, рассматриваемый частный случай задачи линейного программирования может быть решён с помощью алгоритма Беллмана-Форда.

Линейное программирование

Общая *задача линейного программирования* (linear-programming problem) состоит в следующем. Даны $m \times n$-матрица A, m-вектор b и n-вектор c. Требуется найти n-вектор x, являющийся точкой максимума *целевой функции* (objective function) $\sum_{i=1}^{n} c_i x_i$ на множестве, заданном m неравенствами, которые мы можем записать как $Ax \leqslant b$.

Задачи линейного программирования часто возникают в приложениях, поэтому ими много занимались. На практике задачи линейного программирования быстро решаются с помощью *симплекс-метода* (simplex algorithm). Симплекс-метод основан на просмотре вершин многогранника, задаваемого неравенствами-ограничениями $Ax \leqslant b$, при котором значение целевой функции увеличивается. (Симплекс-метод подробно описан в книге Данцига [53].)

Можно, однако, построить последовательность задач линейного программирования, для которой симплекс-метод будет требовать экспоненциального (от размера входа) времени. *Метод эллипсоидов* (ellipsoid algorithm) решает любую задачу линейного программирования за полиномиальное время, но на практике работает медленно. Существует также полиномиальный *алгоритм Кармаркара* (Karmarkar's algoritm), сравнимый по практической эффективности с симплекс-методом.

Мы не будем заниматься алгоритмами для решения общих задач линейного программирования (что требовало бы большего знакомства с линейной алгеброй, чем остальной материал книги), а сделаем только два замечания. Во-первых, если задачу можно свести к задаче линейного программирования с исходными данными полиномиального размера, то эта задача заведомо может быть решена с помощью полиномиального алгоритма. Во-вторых, для многих специальных задач линейного программирования существуют алгоритмы, приводящие к цели быстрее, чем алгоритмы для общего случая. Один пример такого рода доставляет задача, разбираемая в этом разделе; другие примеры — сводящиеся к задачам линейного программирования задача о кратчайшем пути между данной парой вершин (упражнение 25.5-4) и задача о максимальном потоке (упражнение 27.1-8).

В некоторых случаях целевая функция нас не интересует, а требуется найти хотя бы одно *допустимое решение* (feasible solution), то есть решение системы неравенств $Ax \leqslant b$ (или доказать, что таковых нет). Мы займемся именно задачей такого типа.

Системы ограничений на разности

Система ограничений на разности (system of difference constraints) — это система линейных неравенств $Ax \leqslant b$, для которой в каждой строке матрицы A присутствуют ровно два ненулевых элемента, один из которых равен 1, а другой -1. Иными словами, все неравенства имеют вид

$$x_i - x_j \leqslant b_k,$$

где $1 \leqslant i, j \leqslant n$ и $1 \leqslant k \leqslant m$.

Например, задача нахождения 5-вектора $x = (x_i)$, удовлетворяющего условию

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \leqslant \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix},$$

равносильна системе линейных неравенств

$$\begin{aligned} x_1 - x_2 &\leqslant 0, \\ x_1 - x_5 &\leqslant -1, \\ x_2 - x_5 &\leqslant 1, \\ x_3 - x_1 &\leqslant 5, \\ x_4 - x_1 &\leqslant 4, \\ x_4 - x_3 &\leqslant -1, \\ x_5 - x_3 &\leqslant -3, \\ x_5 - x_4 &\leqslant -3. \end{aligned} \tag{25.4}$$

Одно из решений этой системы есть $x = (-5, -3, 0, -1, -4)$. Если ко всем компонентам вектора x прибавить одно и то же число, получится другое решение:

Лемма 25.16

Если $x = (x_1, \ldots, x_n)$ — решение системы ограничений на разности и d — константа, то $x' = (x_1 + d, \ldots, x_n + d)$ — решение той же системы.

Рисунок 25.9 Граф ограничений, соответствующий системе (25.4). В каждой вершине v_i написано число $\delta(v_0, v_i)$. Вектор $x = (-5, -3, 0, -1, -4)$ является одним из решений системы (25.4).

Системы ограничений на разности возникают во многих приложениях. Например, x_i могут быть сроками начала различных работ при строительстве дома. Если x_1 — срок начала рытья котлована под фундамент, а x_2 — срок начала заливки фундамента, и если рытье котлована занимает 3 дня, то $x_1 - x_2 \leqslant -3$.

Графы ограничений

Если система ограничений на разности представлена в виде $Ax \leqslant b$, где A — $m \times n$-матрица, удобно рассмотреть ориентированный граф, для которого A будет матрицей инцидентности (см. упражнение 23.1-7). Точнее говоря, мы сделаем не совсем это, а свяжем с системой взвешенный ориентированный граф $G = (V, E)$, в котором $V = \{v_0, v_1, \ldots, v_n\}$ (по одной вершине на каждое неизвестное плюс вершина v_0), а каждому неравенству $x_j - x_i \leqslant b_k$ соответствует ребро (v_i, v_j) с весом b_k (несколько неравенств с одинаковой левой частью можно заменить одним, взяв минимум их правых частей). Кроме того, в графе имеются n ребер $(v_0, v_1), (v_0, v_2), \ldots, (v_0, v_n)$ (каждое — веса 0). На рис. 25.9 изображен граф, соответствующий системе (25.4).

Следующая теорема показывает, что решения систем разностных ограничений можно находить с помощью алгоритмов поиска кратчайших путей.

Теорема 25.17

Пусть $G = (V, E)$ — граф, соответствующий системе разностных ограничений с n неизвестными. Если он не содержит циклов отрицательного веса, то вектор

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \ldots, \delta(v_0, v_n)) \qquad (25.5)$$

является решением системы. Если G содержит цикл отрицательного веса, то система разностных ограничений несовместна.

Доказательство

Если цикла отрицательного веса нет, то все $\delta(v_0, v_j)$ являются (конечными) числами и лемма 25.3 гарантирует, что числа x_i удовлетворяют всем неравенствам.

Если цикл отрицательного веса есть, то он не содержит вершины v_0, поскольку ни одно ребро в эту вершину не входит, так что все ребра цикла соответствуют каким-то неравенствам системы. Складывая неравенства, соответствующие ребрам цикла, получим (поскольку левые части неравенств цикла сокращаются) неравенство вида $0 \leqslant b$, где $b < 0$ — вес цикла. Следовательно, система несовместна.

Решение систем ограничений на разности

Теорема 25.17 показывает, что найти хотя бы одно решение системы ограничений на разности можно с помощью алгоритма Беллмана-Форда. Заметим, что цикл отрицательного веса, если он есть, обязательно достижим из вершины v_0 (поскольку из v_0 идут стрелки во все остальные вершины), так что система несовместна тогда и только тогда, когда алгоритм Беллмана-Форда возвращает значение FALSE.

Система m разностных ограничений с n неизвестными порождает граф с $n+1$ вершиной и не более чем $n+m$ ребрами. Поэтому найти хотя бы одно решение можно за время $O((n+1)(n+m)) = O(n^2+mn)$. В упражнении 25.5-5 мы попросим вас доказать, что можно модифицировать этот алгоритм таким образом, чтобы он находил решение или устанавливал несовместность системы за время $O(mn)$.

Упражнения

25.5-1 Найдите хотя бы одно решение или установите несовместность следующей системы неравенств:

$$\begin{aligned}
x_1 - x_2 &\leqslant 1,\\
x_1 - x_4 &\leqslant -4,\\
x_2 - x_3 &\leqslant 2,\\
x_2 - x_5 &\leqslant 7,\\
x_2 - x_6 &\leqslant 5,\\
x_3 - x_6 &\leqslant 10,\\
x_4 - x_2 &\leqslant 2,\\
x_5 - x_1 &\leqslant -1,\\
x_5 - x_4 &\leqslant 3,\\
x_6 - x_3 &\leqslant -8.
\end{aligned}$$

25.5-2 То же задание, что в предыдущем упражнении, для системы

$$\begin{aligned}
x_1 - x_2 &\leqslant 4,\\
x_1 - x_5 &\leqslant 5,\\
x_2 - x_4 &\leqslant -6,\\
x_3 - x_2 &\leqslant 1,\\
x_4 - x_1 &\leqslant 3,\\
x_4 - x_3 &\leqslant 5,\\
x_4 - x_5 &\leqslant 10,\\
x_5 - x_3 &\leqslant -4,\\
x_5 - x_4 &\leqslant -8.
\end{aligned}$$

25.5-3 Может ли в графе ограничений кратчайший путь из v_0 в какую-то вершину иметь положительный вес?

25.5-4 Сведите задачу о кратчайшем пути между парой вершин к задаче линейного программирования.

25.5-5 Модифицируйте алгоритм Беллмана-Форда таким образом, чтобы при решении системы m ограничений на разности с n неизвестными он работал за время $O(mn)$.

25.5-6 Как с помощью алгоритма, аналогичного алгоритму Беллмана-Форда, решить систему ограничений на разности, пользуясь графом ограничений без дополнительной вершины v_0?

25.5-7* Покажите, что решение системы разностных ограничений с n неизвестными, находимое алгоритмом Беллмана-Форда, имеет (среди всех решений этой системы, в которых все переменные неположительны) максимальное значение суммы $x_1 + x_2 + \ldots + x_n$.

25.5-8* Покажите, что решение системы разностных ограничений $Ax \leqslant b$ с n неизвестными, находимое алгоритмом Беллмана-Форда, имеет минимально возможное значение величины $\max\{x_i\} - \min\{x_i\}$ среди всех оещений этой системы. Чем полезно это обстоятельство при планировании строительства?

25.5-9 Рассмотрим систему неравенств, в которой каждое неравенство является либо ограничением на разности, либо неравенством вида $x_i \leqslant a$, либо неравенством вида $x_j \geqslant a$. Модифицируйте алгоритм Беллмана-Форда таким образом, чтобы он находил хотя бы одно решение или устанавливал несовместность таких систем.

25.5-10 Пусть к системе ограничений на разности добавлено некоторое количество уравнений вида $x_i = x_j + b_k$. Модифицируйте алгоритм Беллмана-Форда таким образом, чтобы он мог находить решения таких систем.

25.5-11 Разработайте эффективный алгоритм для нахождения решения системы ограничений на разности, в котором все неизвестные являются целыми числами (константы в ограничениях не обязаны быть целыми, но их можно заменить на их целые части).

25.5-12* Разработайте эффективный алгоритм для нахождения решения системы ограничений на разности, если все переменные разбиты на две группы: целые (для которых допустимы только целые значения) и вещественные (для которых такого ограничения нет).

Задачи

25-1 Модификация алгоритма Беллмана-Форда по Йену

Выберем порядок, в котором обрабатываются рёбра в алгоритме Беллмана-Форда, следующим образом. Пронумеруем каким-либо образом вершины графа и разобьем множество E вершин графа на два подмножества: E_f, состоящее из стрелок, идущих из вершины с меньшим номером в вершину с бо́льшим номером, и E_b, состоящее из стрелок, идущих из вершины с бо́льшим номером в вершину с меньшим номером. Пусть $G_f = (V, E_f)$ и $G_b = (V, E_b)$ (через V обозначено множество вершин графа). Начнём с такого очевидного наблюдения:

(а) Покажите, что графы G_f и G_b ацикличны, причём расположение вершин в порядке возрастания (убывания) номеров задает топологическое упорядочение на графе G_f (G_b).

Будем теперь проводить релаксацию ребер при каждой итерации цикла в алгоритме Беллмана-Форда в следующем порядке: сначала перебираем вершины в порядке возрастания номеров и для каждой вершины подвергаем релаксации все выходящие из нее ребра графа E_f; затем перебираем все вершины в порядке убывания номеров и для каждой вершины подвергаем релаксации все выходящие из нее ребра графа E_b.

(б) Пусть G не содержит циклов отрицательного веса, достижимых из вершины s; докажите, что после $\lceil |V|/2 \rceil$ итераций цикла равенства $d[v] = \delta(s, v)$ будут выполняться для всех $v \in V$.

(в) Оцените время работы описанной модификации алгоритма Беллмана-Форда.

25-2 Вложенные ящики

Будем говорить, что d-мерный ящик размеров $(x_1, x_2, \ldots, x_d)$ *вкладывается* (nests) в ящик размеров $(y_1, y_2, \ldots, y_d)$, если у множества $\{1, 2, \ldots, d\}$ существует такая перестановка π, что $x_{\pi(1)} < y_1, x_{\pi(2)} < y_2, \ldots, x_{\pi(d)} < y_d$.

(а) Покажите, что отношение *"вкладываться"* транзитивно.

(б) Опишите эффективный способ проверить, вкладывается ли один d-мерный ящик в другой.

(в) Даны n различных d-мерных ящиков. Требуется узнать, какое максимальное число из них можно последовательно вложить друг в друга (первый во второй, второй в третий и т.д.). Укажите эффективный алгоритм для решения этой задачи и оцените время его работы.

25-3 Арбитражные операции

Арбитражными операциями (arbitrage) называется следующий способ извлекать прибыль из несогласованности курсов обмена валют. Предположим, что один доллар можно обменять на 0,7 фунта стерлингов, один фунт стерлингов — на 9,5 франков, и один франк — на 0,16 доллара. Тогда, обменивая 1 доллар в указанной последовательности, в результате можно получить 1,064 доллара и тем самым остаться с прибылью 6,4%.

Пусть имеются n валют (пронумерованных от 1 до n) и массив $R[1..n, 1..n]$, в котором записаны курсы обмена (единицу валюты i можно обменять на $R[i, j]$ единиц валюты j).

(а) Разработайте эффективный алгоритм, позволяющий выяснить, существует ли такая последовательность $(i_1, i_2, \ldots, i_k)$, что

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdot \cdots \cdot R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1.$$

Оцените время работы вашего алгоритма.

(б) Разработайте эффективный алгоритм, печатающий такую последовательность, если она существует. Оцените время его работы.

25-4 Алгоритм Габоу нахождения кратчайших путей с помощью масштабирования

Пусть нам дан взвешенный ориентированный граф $G = (V, E)$, в котором веса всех ребер являются целыми неотрицательными числами, не превосходящими W. Покажем, как можно найти кратчайшие пути из одной вершины за время $O(E \lg W)$.

Пусть $k = \lceil \lg(W+1) \rceil$ — количество битов в двоичном представлении числа W. Для $i = 1, 2, \ldots, k$ положим $w_i(u,v) = \lfloor w(u,v)/2^{k-i} \rfloor$ (иными словами, $w_i(u,v)$ получается из $w(u,v)$ отбрасыванием $k-i$ младших битов в двоичном представлении числа $w(u,v)$). Например, если $k = 5$ и $w(u,v) = 25 = \langle 11001 \rangle$, то $w_3(u,v) = \langle 110 \rangle = 6$. В частности, w_1 принимает только значения 0 и 1, определяемые старшим разрядом, а $w_k = w$.

Пусть $\delta_i(u,v)$ — вес кратчайшего пути из u в v относительно весовой функции w_i (в частности, $\delta_k(u,v) = \delta(u,v)$). Алгоритм, о котором пойдёт речь в этой задаче, найдёт сначала все $\delta_1(s,v)$ (s — исходная вершина), затем все $\delta_2(s,v)$, и так далее, пока не дойдёт до $\delta_k(s,v) = \delta(s,v)$. Далее мы полагаем, что $|E| \geqslant |V| - 1$; как мы увидим, стоимость нахождения δ_i при известном δ_{i-1} есть $O(E)$, так что алгоритм будет работать за время $O(kE) = O(E \lg W)$.

Такой план решения задачи — замена исходных данных их двоичными приближениями с последовательным уточнением — называется *масштабированием* (scaling)

(а) Пусть $\delta(s,v) \leqslant |E|$ для всех вершин $v \in V$ (предполагается, что $|E| \geqslant |V| - 1$ и веса являются целыми неотрицательными числами). Покажите, что можно найти $\delta(s,v)$ для всех $v \in V$ за время $O(E)$.

(б) Покажите, что можно подсчитать $\delta_1(s,v)$ для всех $v \in V$ за время $O(E)$.

Теперь займемся вычислением δ_i исходя из δ_{i-1}.

(в) Докажите, что (при $i = 2, 3, \ldots, k$) либо $w_i(u,v) = 2w_{i-1}(u,v)$, либо $w_i(u,v) = 2w_{i-1}(u,v) + 1$. Выведите отсюда, что

$$2\delta_{i-1}(u,v) \leqslant \delta_i(u,v) \leqslant 2\delta_{i-1}(u,v) + |V| - 1$$

для всех $v \in V$.

(г) Для $i = 2, 3, \ldots, k$ и $(u,v) \in E$ положим

$$\hat{w}_i(u,v) = w_i(u,v) + 2\delta_{i-1}(s,u) - 2\delta_{i-1}(s,v).$$

Покажите, что $\hat{w}_i(u,v) \geqslant 0$.

(д) Пусть $\hat{\delta}_i(s,v)$ — вес кратчайшего пути относительно весовой функции $\hat{w}_i$. Покажите, что

$$\delta_i(s,v) = \hat{\delta}_i(s,v) + 2\delta_{i-1}(s,v)$$

и что $\hat{\delta}_i(s,v) \leqslant |E|$.

(е) Объясните, как за время $O(E)$ вычислить все значения $\delta_i(s,v)$, зная $\delta_{i-1}(s,v)$. Как вычислить $\delta(s,v)$ (для всех $v \in V$) за время $O(E \lg W)$?

25-5 Алгоритм Карпа для отыскания цикла с минимальным средним весом

Пусть $G = (V,E)$ — ориентированный граф с весовой функцией $w\colon E \to \mathbb{R}$, и пусть $n = |V|$. *Средним весом* (mean weight) цикла $c = \langle e_1, e_2, \ldots, e_k \rangle$, где e_j — ребра графа, назовем число

$$\mu(c) = \frac{1}{k} \sum_{i=1}^{k} w(e_i).$$

Пусть $\mu^* = \min_c \mu(c)$, где c пробегает все (ориентированные) циклы. В этой задаче мы опишем эффективный алгоритм для вычисления μ^*.

Не ограничивая общности, будем считать, что каждая вершина $v \in V$ достижима из некоторой вершины s. Через $\delta(s,v)$ обозначим вес кратчайшего пути из s в v; пусть $\delta_k(s,v)$ — вес кратчайшего пути из s в v, состоящего в точности из k рёбер (если такого пути нет, полагаем $\delta_k(s,v) = \infty$).

(а) Пусть $\mu^* = 0$. Покажите, что G не содержит циклов отрицательного веса и что $\delta(s,v) = \min_{0 \leqslant k \leqslant n-1} \delta_k(s,v)$ для всех $v \in V$.

(б) Пусть $\mu^* = 0$. Покажите, что

$$\max_{0 \leqslant k \leqslant n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} \geqslant 0$$

для любой вершины $v \in V$ (Указание: воспользуйтесь двумя утверждениями предыдущего пункта.)

(в) Пусть u и v — две вершины, лежащие на цикле нулевого веса. Пусть вес участка этого цикла от u до v равен x. Покажите, что $\delta(s,v) = \delta(s,u) + x$ (Указание: вес участка от v до u равен $-x$).

(г) Пусть $\mu^* = 0$. Покажите, что существует вершина v, лежащая на цикле с минимальным средним весом, такая, что

$$\max_{0 \leqslant k \leqslant n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} = 0$$

(Указание: покажите, что кратчайший путь от s до вершины, лежащей на цикле с нулевым весом, можно продолжить вдоль этого цикла так, что он останется кратчайшим.)

(д) Пусть $\mu^* = 0$. Покажите, что

$$\min_{v \in V} \max_{0 \leqslant k \leqslant n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} = 0.$$

(е) Покажите, что

$$\mu^* = \min_{v \in V} \max_{0 \leqslant k \leqslant n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k}$$

(Указание: если прибавить константу t к весам всех ребер, то μ^* увеличится на t.)

(ж) Разработайте алгоритм, вычисляющий μ^* за время $O(VE)$.

Замечания

Алгоритм Дейкстры [55] появился в 1959 году (без упоминания очередей с приоритетами). Алгоритм Беллмана-Форда основан на двух отдельных алгоритмах, изобретённых Беллманом [22] и Фордом [71]. Связь кратчайших путей с ограничениями на разности описана Беллманом. Алгоритм для поиска кратчайших путей в ациклическом ориентированном графе за линейное время описан Лоулером [132] (как *"фольклорный"*).

Если веса рёбер — небольшие целые числа, то для нахождения кратчайших путей из одной вершины можно применить и более эффективные методы. Ахуджа, Мельхорн, Орлин и Тарьян [6] описывают алгоритм, работающий за время $O(E+V\sqrt{\lg W})$ в предположении, что веса — целые неотрицательные числа, не превосходящие W. Они же приводят простой алгоритм, работающий за время $O(E+V\lg W)$. Для случая, когда веса могут быть отрицательными (целыми) числами, есть алгоритм Габоу и Тарьяна [77], работающий за время $O(\sqrt{V}E\lg(VW))$, где W — максимум абсолютных величин весов.

Хороший обзор различных алгоритмов, связанных с линейным программированием (в частности, симплекс-метода и метода эллипсоидов), дают Пападимитриу и Стайглиц [154]. Симплекс-метод был изобретён Данцигом (G. Dantzig) в 1947 году, и различные варианты этого метода до сих пор остаются наиболее популярными способами решения задач линейного программирования. Метод эллипсоидов предложил Л.Г.Хачиян, основываясь на работах Н.З. Шора, Д.Б. Юдина и А.С. Немировского. Кармаркар описывает свой алгоритм в [115].

26 Кратчайшие пути для всех пар вершин

В этой главе мы займёмся задачей о нахождении кратчайших путей для всех пар вершин графа. Таблица расстояний между всевозможными парами городов в атласе дорог получается в результате решения именно такой задачи. Как и в главе 25, будем рассматривать ориентированный взвешенный граф $G = (V, E)$ с вещественной весовой функцией $w\colon E \to \mathbb{R}$. Для каждой пары вершин $u, v \in V$ мы должны найти кратчайший путь u в v, точнее, путь наименьшей длины (длина пути определяется как сумма весов всех его рёбер). Таким образом, ответом в задаче о кратчайших путях можно считать таблицу, в которой на пересечении строки u и столбца v которой находится вес кратчайшего пути из u в v (дополненную некоторой информацией о самих этих путях, см. ниже).

Разумеется, можно решить эту задачу, если $|V|$ раз применить алгоритм для поиска кратчайших путей из одной вершины (ко всем вершинам графа по очереди). Если все ребра графа имеют неотрицательные веса, то разумно использовать алгоритм Дейкстры; при простой реализации очереди с приоритетами с помощью массива общее время работы алгоритма составит $O(V^3 + VE) = O(V^3)$.

Если очередь реализовать с помощью двоичной кучи, то общая стоимость составит $O(VE \lg V)$, что даёт выигрыш для разреженных графов. Обе эти оценки можно улучшить, использовав фибоначчиевы кучи, для которых общее время работы алгоритма есть алгоритма будет $O(V^2 \lg V + VE)$.

Если в графе есть рёбра с отрицательными весами, то алгоритм Дейкстры применить нельзя. Если вместо него использовать более медленный алгоритм Беллмана-Форда, выполняя его для каждой вершины графа, общее время работы составит $O(V^2 E)$ — для плотных графах это будет $O(V^4)$. Алгоритмы, описанные ниже, работают быстрее. Кроме того, в этой главе мы установим связь между задачей о нахождении кратчайших путей для всех пар вершин графа и умножением матриц, а также исследуем ее алгебраическую структуру.

В большинстве алгоритмов этой главы графы представляются матрицами смежности. Исключением является алгоритм Джонсона для разреженных графов, который (как и алгоритмы поиска из одной вершины) использует списки смежности. Естественно сочетать сведения о наличии ребра и о его весе в одной матрице, полагая веса отсутствующих рёбер бесконечными.

Таким образом, при обработке взвешенного ориентированного графа $G = (V, E)$ алгоритму даётся матрица $W = (w_{ij})$, где

$$w_{ij} = \begin{cases} 0 & \text{если } i = j, \\ \text{вес (ориентированного) ребра } (i,j) & \text{если } i \neq j \text{ и } (i,j) \in E, \\ \infty & \text{если } i \neq j \text{ и } (i,j) \notin E. \end{cases} \tag{26.1}$$

Ребра могут иметь отрицательный вес. Мы будем, однако, предполагать, что циклов отрицательного веса в графе нет.

Представленные в данной главе алгоритмы нахождения кратчайших путей для всех пар вершин будут вычислять матрицу $D = (d_{ij})$ размером $n \times n$, элемент d_{ij} которой содержит вес кратчайшего пути из вершины i в вершину j, то есть равен $\delta(i, j)$ в обозначениях предыдущей главы.

Решение задачи о кратчайших путях для всех пар вершин должно включать в себя не только веса кратчайших путей, но и *матрицу предшествования* (predecessor matrix) $\Pi = (\pi_{ij})$, в которой π_{ij} является вершиной, предшествующей j на одном из кратчайших путей из i в j. (мы полагаем $\pi_{ij} = \textsc{nil}$, если $i = j$ или путей из i в j не существует). Для каждой вершины $i \in V$ можно определить *подграф предшествования* (predecessor subgraph) $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$, где

$$V_{\pi,i} = \{j \in V : \pi_{ij} \neq \textsc{nil}\} \cup \{i\},$$

и

$$E_{\pi,i} = \{(\pi_{ij}, j) : j \in V_{\pi,i} \text{ и } \pi_{ij} \neq \textsc{nil}\}.$$

Мы будем требовать, чтобы для каждого i подграф предшествования $G_{\pi,i}$ был деревом кратчайших путей из вершины i (в смысле главы 25). В этом случае следующая процедура печатает кратчайший путь из вершины i в вершину j.

```
{\sc Print-All-Pairs-Shortest-Path}$(\Pi,i,j)$\\
if $i=j$
then print $i$
else if $\pi_{ij}=\mbox{\sc nil}$
    then print ``Пути из'' $i$ ``в'' $j$ ``нет''
    else {\sc Print-All-Pairs-Shortest-Path}$(\Pi,i,\pi_{ij})$
        print $j$
```

Мы не будем подробно говорить о построении матрицы предшествования (её свойствам посвящено несколько упражнений).

План главы

В разделе 26.1 рассматривается алгоритм решения задачи о кратчайших расстояниях для всех пар вершин, в основе которого лежит умножение матриц. Время работы этого алгоритма — $\Theta(V^3 \lg V)$; он вычисляет степень матрицы, многократно возводя её в квадрат.

Более быстрый ($O(V^3)$) алгоритм Флойда–Уоршолла, также использующий технику динамического программирования, излагается в разделе 26.2. В этом же разделе рассматривается задача о транзитивном замыкании ориентированного графа, которая оказывается тесно связанной с задачей нахождения кратчайших путей для всех пар вершин.

В разделе 26.3 излагается алгоритм Джонсона. В отличие от других алгоритмов этой главы, он использует в своей работе списки смежных вершин, а не матрицу смежности. Время работы этого алгоритма есть $O(V^2 \lg V + VE)$; таким образом, он эффективен для разреженных графов.

В последнем разделе (26.4) мы исследуем алгебраическую структуру (замкнутое полукольцо), позволяющую применить алгоритмы нахождения кратчайших путей для решения других задач с графами, в которых также требуется определить что-либо для всех пар вершин.

Всюду в этой главе мы рассматриваем граф $G = (V, E)$ с n , вершинами, так что $|V| = n$. Мы будем обозначать матрицы большими буквами (например, W или D) а отдельные элементы матриц соответствующими маленькими буквами (w_{ij}, d_{ij}). Кроме того, мы будем использовать верхний индекс в скобках (напродобие $D^{(m)} = (d_{ij}^{(m)})$), роль которого будет аналогична степени матрицы (см. ниже). Наконец, размер n квадратной $n \times n$-матрицы A мы будем иногда записывать как $rows[A]$ (rows — строки).

26.1 Кратчайшие пути и умножение матриц

В этом разделе мы рассмотрим алгоритм динамического программирования решения задачи нахождения кратчайших путей для всех пар вершин ориентированного графа $G = (V, E)$. На каждом шаге он будет почти что умножать матрицы — только не вполне обычным образом. Сначала мы построим алгоритм со сложностью (временем работы) $\Theta(V^4)$, а затем улучшим его, получив оценку $\Theta(V^3 \lg V)$. Наше изложение будет следовать схеме алгоритма динамического программирования (глава 16)

Структура кратчайшего пути

Убедимся, что части решения являюётся решениями аналогичных подзадач. В нашем случае это означает, что отрезки кратчайшего пути сами являются кратчайшими путями между соответствующими вершинами (лемма 25.1).

Рекурсивная формула для длины кратчайшего пути

Пусть $d_{ij}^{(m)}$ обозначает минимальный вес пути из вершины i в вершину j, если рассматривать пути с не более чем m рёбрами.

При $m = 0$ допустимы лишь ”*пути*” вовсе без рёбер, то есть

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{если } i = j, \\ \infty & \text{если } i \neq j. \end{cases}$$

Если же $m \geqslant 1$, то минимум $d_{ij}^{(m)}$ достигается либо на пути из не более чем $m-1$ ребра (и тогда равен $d_{ij}^{(m-1)}$), либо же на пути из m ребер. В последнем случае этот путь можно разбить на начальный отрезок из $m-1$ рёбер, ведущий из начальной вершины i в некоторую вершину k, и на последнее ребро (k,j). Мы приходим к формуле

$$d_{ij}^{(m)} = \min(d_{ij}^{(m-1)}, \min_{1\leqslant k\leqslant n}\{d_{ik}^{(m-1)} + w_{kj}\}) =$$

$$= \min_{1\leqslant k\leqslant n}\{d_{ik}^{(m-1)} + w_{kj}\}$$

(Последнее равенство использует равенство $w_{jj} = 0$.)

Если граф не содержит циклов с отрицательными весами, кратчайший путь можно выбрать без циклов; такой путь содержит не более $n-1$ рёбер. Следовательно,

$$\delta(i,j) = d_{ij}^{(n-1)} = d_{ij}^{(n)} = d_{ij}^{(n+1)} = \cdots. \tag{26.3}$$

Вычисление кратчайших путей ”*снизу вверх*”

По заданной матрице весов $W = (w_{ij})$ мы будем последовательно вычислять матрицы $D^{(1)}, D^{(2)}, \ldots, D^{(n-1)}$, где $D^{(m)} = (d_{ij}^{(m)})$. Как мы видели, последняя матрица $D^{(n-1)}$ будет содержать веса кратчайших путей. Заметим, что матрица $D^{(1)}$ (веса путей из одного ребра) совпадает с W.

Шаг алгоритма состоит в вычислении $D^{(m)}$ по $D^{(m-1)}$ и W.

```
{\sc Extend-Shortest-Paths}$(D,W)$\\
\verb|1 |$n \leftarrow rows[D]$\\
\verb|2 |пусть $D'=(d'_{ij})$ --- $n\times n$-матрица
\verb|3 |for $i \leftarrow 1$ to $n$\\
\verb|4     |do for $j \leftarrow 1$ to $n$\\
\verb|5            |do $d'_{ij} \leftarrow \infty$\\
\verb|6               |for $k \leftarrow 1$ to $n$\\
\verb|7                  |do $d'_{ij} \leftarrow \min(d'_{ij},\,d_{ik}+w_{
\verb|8 |return $D'$\\
```

Эта процедура вычисляет матрицу D' в соответствии с формулой (26.2) при этом роль матриц D и D' играют матрицы $D^{(m-1)}$ и $D^{(m)}$. Процедура содержит три вложенных цикла, так что время её работы есть $\Theta(n^3)$.

Чтобы увидеть аналогию этой процедуры с процессом умножения матриц, запишем процедуру умножения матриц A и B размером $n \times n$ по формуле

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \qquad (26.4)$$

```
{\sc Matrix-Multiply}$(A,B)$\\
\verb|1 |$n \leftarrow rows[A]$\\
\verb|2 |пусть $C=(c_{ij})$ --- $n\times n$-матрица
\verb|3 |for $i \leftarrow 1$ to $n$\\
\verb|4     |do for $j \leftarrow 1$ to $n$\\
\verb|5            |do $c_{ij} \leftarrow 0$\\
\verb|6               |for $k \leftarrow 1$ to $n$\\
\verb|7                   |do $c_{ij} \leftarrow c_{ij}+a_{ik}\cdot
b_{kj}$\\
\verb|8 |return $C$
```

Видно, что эта процедура получается из предыдущей заменами

$$\begin{array}{rcl} d^{(m-1)} & \to & a, \\ w & \to & b, \\ d^{(m)} & \to & c, \\ \min & \to & +, \\ + & \to & \cdot \end{array}$$

При этом символу ∞, являющемуся нейтральным элементом для операции min (в том смысле, что $\min(\infty, a) = a$, соответствует число 0, являющееся нейтральным элементов для операции $+$ $(0+a = a)$.

С точки зрения этой аналогии, мы как бы вычисляем " *произведение*" $n-1$ экземпляров матрицы W с помощью последовательных умножений:

$$\begin{array}{lclcl} D^{(1)} & = & D^{(0)} \cdot W & = & W, \\ D^{(2)} & = & D^{(1)} \cdot W & = & W^2, \\ D^{(3)} & = & D^{(2)} \cdot W & = & W^3, \\ & & \vdots & & \\ D^{(n-1)} & = & D^{(n-2)} \cdot W & = & W^{n-1}. \end{array}$$

Результат этих "*умножений*", матрица $D^{(n-1)} = W^{n-1}$ содержит веса кратчайших путей. Оформим описанное вычисление в виде процедуры (с временем работы $\Theta(n^4)$):

```
{\sc Slow-All-Paths-Shortest-Paths}$(W)$\\
\verb|1 |$n \leftarrow rows[W]$\\
\verb|2 |$D^{(1)} \leftarrow W$\\
```

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad D^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Рисунок 26.1 26.1 Ориентированный граф G и последовательность матриц $D^{(m)}$. Можно убедиться, что матрица $D^{(5)} = D^{(4)} \cdot W$ (а значит, и все последующие) будет равна $D^{(4)}$.

```
\verb|3 |for $m \leftarrow 2$ to $n-1$\\
\verb|4     |do $D^{(m)} \leftarrow \mbox{\sc Extend-
Shortest-Paths}(D^{(m-1)},W
)$\\
\verb|5 |return $D^{(n-1)}$\\
```

На рис. 26.1 показан пример графа и соответствующих ему матриц $D^{(m)}$.

Более быстрый способ

Заметим, что мы вычисляем все матрицы $D^{(m)}$, хотя нас интересует лишь матрица $D^{(n-1)}$ или любая из следующих за ней (при отсутствии циклов с отрицательными весами все они равны). Продолжим аналогию с умножением: степень числа a можно вычислить быстрее, если не домножать всё время на a, а возводить в квадрат (такой метод заведомо применим, если показатель степени есть $2, 4, 8, \ldots$).

Аналогичным образом мы мы можем определить $D^{(n-1)}$, выполнив всего $\lceil \lg(n-1) \rceil$ умножений матриц, вычисляя матрицы в последовательности

$$\begin{array}{lllll} D^{(1)} & = & W & & \\ D^{(2)} & = & W^2 & = & W \cdot W, \\ D^{(4)} & = & W^4 & = & W^2 \cdot W^2, \\ D^{(8)} & = & W^8 & = & W^4 \cdot W^4, \end{array}$$

то тех пор, пока показатель степени станет большим или равным $n-1$ (при этом он будет равен $2^{\lceil \lg(n-1) \rceil}$, как легко видеть).

Реализуем этот метод *повторного возведения в квадрат* (repeated squaring) в виде процедуры:

```
{\sc Faster-All-Pairs-Shortest-Paths}$(W)$\\
```

Рисунок 26.2 26.2 Взвешенный ориентированный граф, используемый в упр. 26.1-1, 26.2-1 и 26.3-1.

```
\verb|1 |$n \leftarrow rows[W]$\\
\verb|2 |$D^{(1)} \leftarrow W$\\
\verb|3 |$m \leftarrow 1$\\
\verb|4 |while $n-1 > m$\\
\verb|5      |do $D^{(2m)} \leftarrow \mbox{\sc Extend-
Shortest-Paths}(D^{(m)},
D^{(m)})$\\
\verb|6         |$m \leftarrow 2m$\\
\verb|7 |return $D^{(m)}$
```

На каждом шаге цикла **while** в строках 4-6 вычисляется $D^{(2m)} = (D^{(m)})^2$ и значение m удваивается, пока m не станет большим или равным $n-1$.

Время работы такой процедуры составляет $\Theta(n^3 \lg n)$, т.к. всего выполняется $\lceil \lg(n-1) \rceil$ *"умножений"* матриц, каждое из которых требует $\Theta(n^3)$ действий. Алгоритм довольно прост, и можно надеяться, что константа, скрытая в Θ-обозначени, будет небольшой.

[Заметим, что мы неявно использовали ассоциативность нашего *" умножения"* матриц, переходя к другому способу вычисления степеней. Её можно проверить по аналогии с доказательством ассоциативности для обычного умножения — либо заменить ссылку на ассоциативность прямым доказательством того, что $D^{(2m)}$ есть произведение $D^{(m)}$ на $D^{(m)}$, которое легко провести, рассматривая две половины пути длины $2m$.]

Упражнения

26.1-1 Проследите за исполнением алгоритмов Slow-All-Pairs-Shortest-Paths и Faster-All-Pairs-Shortest-Paths на графе рис. 26.2. Вычислите все возникающие при этом матрицы.

26.1-2 Где используется, что $w_{ii} = 0$ при всех i?

26.1-3 Что соответствует матрице

$$D^{(0)} = \begin{pmatrix} 0 & \infty & \infty & \ldots & \infty \\ \infty & 0 & \infty & \ldots & \infty \\ \infty & \infty & 0 & \ldots & \infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \infty & \ldots & 0 \end{pmatrix}$$

если продолжить аналогию с обычным умножением матриц?

26.1-4 Представьте задачу поиска кратчайших путей из одной вершины как задачу отыскания произведений и вектора. Как в этих терминах выглядит алгоритм Беллмана-Форда (см. раздел 25.3)?

26.1-5 Придумайте алгоритм вычисления матрицы предшествования Π по уже имеющейся матрице D весов кратчайших путей за время $O(n^3)$.

26.1-6 С другой стороны, матрицы предшествования могут вычисляться параллельно с вычислением весов. Будем брать в качестве $\pi_{ij}^{(m)}$ предшествующую j вершину на каком-нибудь кратчайшем пути из i в j, состоящем не более чем из m ребер. Измените процедуры EXTEND-SHORTEST-PATHS и SLOW-ALL-PAIRS-SHORTEST-PATHS так, чтобы они в дополнение к $D^{(1)}$, $D^{(2)}$, ..., $D^{(n-1)}$ вычисляли ещё и матрицы $\Pi^{(1)}$, $\Pi^{(2)}$, ..., $\Pi^{(n-1)}$.

26.1-7 Процедура FASTER-ALL-PAIRS-SHORTEST-PATHS (в её нынешнем виде) использует $\lceil \lg(n-1) \rceil$ матриц (массивов) размером $n \times n$. Общий объем памяти, таким образом, составляет $\Theta(n^2 \lg n)$. Измените её так, чтобы использовать всего два массива размером $n \times n$ (тем самым уменьшив объём памяти до $\Theta(n^2)$).

26.7-8 Измените алгоритм FASTER-ALL-PAIRS-SHORTEST-PATHS так, чтобы он обнаруживал наличие в графе цикла с отрицательным весом.

26.1-9 Придумайте эффективный алгоритм определения минимального числа рёбер в цикле отрицательного веса (для данного графа; предполагается, что такой цикл существует).

26.1 Алгоритм Флойда-Уоршолла

В этом разделе мы рассмотрим другой способ решения задачи о кратчайших путях для всех пар вершин ориентированного взвешенного графа, также основанный на технике динамического программирования. Этот способ (алгоритм Флойда–Уоршолла) работает за время $\Theta(V^3)$. Мы по-прежнему допускаем рёбра с отрицательным весом, но запрещаем циклы отрицательного веса.

Кроме того, мы рассмотрим алгоритм нахождения транзитивного замыкания графа, основанный на той же идее.

Строение кратчайшего пути

Алгоритмы предыдущего раздела выделяли последнее ребро пути. Алгоритм Флойда–Уоршалла действует иначе: для него важно, какие вершины используются в качестве промежуточных, и особую роль играют промежуточные вершины с максимальным номером (в некоторой нумерации вершин). *Промежуточной* (intermediate) вершиной простого пути $p = \langle v_1, v_2, \dots v_l \rangle$ будем называть любую из вершин $v_2, v_3, \dots, v_{l-1}$.

Будем считать, что вершинами графа G являются числа $1, 2, \dots, n$. Рассмотрим произвольное $k \leqslant n$. Для данной пары вершин $i, j \in V$ рассмотрим все пути из i в j, у которых все промежуточные вершины принадлежат множеству $\{1, 2, \dots, k\}$. Пусть p — путь минимального веса среди всех таких путей. Он будет простым, так как в графе нет циклов с отрицательным весом. Как найти вес этого пути, зная веса всех таких путей (для всех пар вершин) для меньших k?

Для пути p есть две возможности.

Если k не входит в p, все промежуточные вершины пути p содержатся в множестве $\{1, 2, \ldots, k-1\}$. Тогда путь p является кратчайшим путём из i в j, промежуточные вершины которого принадлежат множеству $\{1, 2, \ldots, k-1\}$.

Если k является промежуточной вершиной пути p, она разбивает его на два участка p_1 и p_2 (вершина k встречается лишь однажды, так как p — простой путь), см. рис. 26.3. По лемме 25.1 путь p_1 будет кратчайшим путём из i в k с промежуточными вершинами из множества $\{1, 2, \ldots, k-1\}$. Путь p_2 является кратчайшим путём из k в j с промежуточными вершинами из множества $\{1, 2, \ldots, k-1\}$.

Рекурентная формула для длин кратчайших путей

Эти рассуждения позволяют написать другую (по сравнению с приведённой в разделе 26.1) рекурентную формулу для длин кратчайших путей. Обозначим через $d_{ij}^{(k)}$ вес кратчайшего пути из вершины i в вершину j с промежуточными вершинами из множества $\{1, 2, \ldots, k\}$. При $k = 0$ промежуточных вершин нет вовсе, поэтому $d_{ij}^{(0)} = w_{ij}$. В общем случае

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{если } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{если } k \geqslant 1. \end{cases} \qquad (26.5)$$

Матрица $D^{(n)} = (d_{ij}^{(n)})$ содержит искомое решение. Другими словами, $d_{ij}^{(n)} = \delta(i, j)$ для всех $i, j \in V$, поскольку разрешены любые промежуточные вершины.

Вычисление кратчайших путей снизу вверх

Напишем процедуру, которая вычисляет веса кратчайших путей, последовательно находя значения $d_{ij}^{(k)}$ для $k = 1, 2, \ldots, n$. Её входм является матрица W размером $n \times n$ (веса рёбер графа), результатом является матрица $D^{(n)}$ весов кратчайших путей.

```
{\sc Floyd-Warshall}$(W)$\\
\verb|1 |$n \leftarrow rows[W]$\\
\verb|2 |$D^{(0)} \leftarrow W$\\
\verb|3 |for $k \leftarrow 1$ to $n$\\
\verb|4    |do for $i \leftarrow 1$ to $n$\\
\verb|5           |do for $j \leftarrow 1$ to $n$\\
\verb|6                  |$d^{(k)}_{ij} \leftarrow \min(d^{(k-
1)}_{ij},
d^{(k-1)}_{ik}+d^{(k-1)}_{kj})$\\
\verb|7 |return $D^{(n)}$
```

На рисунке 26.4 показан ориентированный раф и матрицы $D^{(k)}$, вычисляемые этим алгоритмом.

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Рисунок 26.3 26.4 Матрицы $\Pi^{(k)}$ и $D^{(k)}$, вычисляемые алгоритмом Флойда–Уоршолла для графа рис. 26.1.

Алгоритм Флойда-Уоршолла содержит три вложенных цикла (строки 3–6); время его работы есть $\Theta(n^3)$. Константа, скрытая в Θ-обозначении, невелика, поскольку алгоритм прост и не использует сложных структур данных, так что он применим для достаточно больших графов.

Построение кратчайших путей

Помимо весов кратчайших путей, нас интересуют и сами пути. Один из способов их построения таков: вычислиы вычислении матрицу D их весов, можно затем построить по ней матрицу предшествования Π за время $O(n^3)$ времени (упр. 26.1-5). Затем при помощи функции PRINT-ALL-PAIRS-SHORTEST-PATH можно напечатать кратчайший путь для любой пары вершин.

Другой способ состоит в том, чтобы вычислять матрицу предшествования параллельно с исполнением алгоритма Флойда–Уоршолла. При этом мы вычисляем последовательность матриц $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$, где $\Pi = \Pi^{(n)}$, а $\pi_{ij}^{(k)}$ определяется как вершина, предшествующая вершине j на кратчайшем пути из вершины i в вершину j с промежуточными вершинами из множества $\{1, 2, \ldots, k\}$.

Напишем рекурентную формулу для $\pi_{ij}^{(k)}$. Если $k = 0$, то промежуточных вершин нет, поэтому

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{если } i = j \text{ или } w_{ij} = \infty, \\ i & \text{если } i \neq j \text{ и } w_{ij} < \infty. \end{cases} \tag{26.6}$$

Пусть теперь $k \geqslant 1$. Если кратчайший пути из i в j проходит через вершину k, то предпоследней его вершиной будет та же самая вершина, которая будет предпоследней на кратчайшем пути из k в j с промежуточными вершинами из множества $\{1, 2, \ldots, k-1\}$. Если же путь не проходит через k, то он совпадает с кратчайшим путем из i в j с промежуточными вершинами из множества $\{1, 2, \ldots, k-1\}$. Таким образом,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{если } d_{ij}^{(k-1)} \leqslant d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{если } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases} \tag{26.7}$$

Вычисления по этим формулам легко добавить к алгоритму Флойда–Уоршолла (упр. 26.2-3). На рис. 26.4 показана последовательность матриц $\Pi^{(k)}$, получающаяся в процессе вычислений. В том же упражнении предлагается доказать, что подграф предшествования $G_{\pi,i}$ является деревом кратчайших путей из вершины i. Другой способ построения кратчайших путей указан в упр. 26.2-6.

Транзитивное замыкание ориентированного графа

Задача о транзитивном замыкании состоит в следующем. Дан ориентированный граф $G = (V, E)$ с вершинами $1, 2, \ldots, n$. Требуется определить для любой пары его вершин $i, j \in V$, существует ли в графе путь из вершины i в вершину j. *Транзитивным замыканием* ориентированного графа G называется граф $G^* = (V, E^*)$, где

$$E^* = \{(i,j) : \text{в графе } G \text{ существует путь из } i \text{ в } j\}.$$

Транзитивное замыкание графа можно вычислить за время $\Theta(n^3)$ при помощи алгоритма Флойда–Уоршолла, считая, что все рёбра графа имеют вес 1: если существует путь из вершины i в вершину j, то d_{ij} будет меньеш n, в противном случае $d_{ij} = \infty$.

На практике более выгодно (в смысле времени и памяти) пользоваться несколько другим способом вычисления транзитивного замыкания за время $\Theta(n^3)$. Заменим в алгоритме Флойда–Уоршолла арифметические операции min и + на логические операции $\vee$ и $\wedge$. Другими словами, положим $t_{ij}^{(k)}$ равным 1, если в графе G существует путь из i в j с промежуточными вершинами из множества $\{1, 2, \ldots, k\}$, и равным 0, если такого пути нет. Ребро (i,j) принадлежит транзитивному замыканию G^* тогда и только тогда, когда $t_{ij}^{(n)} = 1$. По аналогии с формулой (26.5) напишем соотношения для $t_{ij}^{(k)}$:

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{если } i \neq j \text{ и } (i,j) \notin E, \\ 1 & \text{если } i = j \text{ или } (i,j) \in E, \end{cases}$$

и (при $k \geqslant 1$)

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}). \tag{26.8}$$

Основанный на этом соотношении алгоритм последовательно вычисляет матрицы $T^{(k)} = (t_{ij}^{(k)})$ для $k = 1, 2, \ldots, n$:

```
{\sc Transitive-Clusure}$(G)$\\
\verb|1 |$n \leftarrow |V[G]|$\\
\verb|2 |for $i\leftarrow 1$ to $n$\\
\verb|3    |do for $j\leftarrow 1$ to $n$\\
\verb|4          |do if $i=j$ or $(i,j)\in E[G]$\\
\verb|5               |then $t^{(0)}_{ij} \leftarrow 1$\\
\verb|6               |else $t^{(0)}_{ij} \leftarrow 0$\\
\verb|7 |for $k\leftarrow 1$ to $n$\\
\verb|8    |do for $i\leftarrow 1$ to $n$\\
\verb|9          |do for $j\leftarrow 1$ to $n$\\
\verb|10               |do $t^{(k)}_{ij}\leftarrow t^{(k-
1)}_{ij}\vee
(t^{(k-1)}_{ik}\wedge t^{(k-1)}_{kj})$\\
\verb|11 |return $T^{(n)}$
```

На рис. 26.5 приведён пример графа и матриц $T^{(k)}$, вычисленных процедурой Transitive-Clusure.

Время работы процедуры Transitive-Clusure составляет $\Theta(n^3)$, как и у алгоритма Флойда–Уоршолла. Однако на многих

$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Рисунок 26.4 26.5 Ориентированный граф и матрицы $T^{(k)}$, вычисленные алгоритмом построения транзитивного замыкания.

компьютерах логические операции выполняются быстрее, чем арифметические операции с целыми числами, поэтому процедура Transitive-Clusure эффективнее алгоритма Флойда-Уоршолла. Кроме того, использование булевских переменных вместо целых сокращает объём используемой памяти.

В разделе 26.4 мы увидим, что аналогия между алгоритмами Флойда–Уоршолла и построением транзитивного замыкания не случайна: оба алгоритма основаны на алгебраической структуре, называемой "*замкнутое полукольцо*".

Упражнения

26.2-1 Исполните алгоритм Флойда-Уоршолла для взвешенного ориентированного графа рис. 26.2, найдя все матрицы $D^{(k)}$.

26.2-2 В приведённом нами виде алгоритм Флойда-Уоршолла требует $\Theta(n^3)$ памяти для хранения матриц $D^{(k)} = (d_{ij}^{(k)})$. Конечно, можно сэкономить место, если хранить только две соседние матрицы. Оказывается, можно пойти ещё дальше: докажите, что если в процедуре Floyd-Warshall убрать верхние индексы, то она по-прежнему будет вычислять веса кратчайших путей.

```
{\sc Floyd-Warshall'}$(W)$\\
\verb|1 |$n \leftarrow rows[W]$\\
\verb|2 |$D \leftarrow W$\\
\verb|3 |for $k \leftarrow 1$ to $n$\\
\verb|4     |do for $i \leftarrow 1$ to $n$\\
\verb|5         |do for $j \leftarrow 1$ to $n$\\
\verb|6             |$d_{ij} \leftarrow \min(d_{ij},
d_{ik}+d_{kj})$\\
\verb|7 |return $D$
```

Тем самым мы сократили объём требуемой памяти до $\Theta(n^2)$.

26.2-3 Добавьте в процедуру Floyd-Warshall вычисление матриц предшествования $\Pi^{(k)}$ по формулам (26.6) и (26.7). Докажите, что для любой вершины $i \in V$ подграф предшествования $G_{\pi,i}$ является деревом кратчайших путей из вершины i.

26.2-4 Будет ли правильно вычислена матрица предшествования

П, если изменить формулу (26.7) так:

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{если } d_{ij}^{(k-1)} < d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{если } d_{ij}^{(k-1)} \geqslant d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

26.2-5 Как можно использовать результат работы алгоритм Флойда–Уоршолла, чтобы узнать, есть ли в графе цикл отрицательного веса?

26.2-6 Еще один способ построения кратчайших путей в алгоритме Флойда-Уоршолла таков. Пусть $\varphi_{ij}^{(k)}$ $(i, j, k = 1, 2, \ldots, n)$ есть промежуточная вершина с максимальным номером на кратчайшем пути из i в j с промежуточными вершинами из множества $\{1, 2, \ldots, k\}$. Напишите рекурентное соотношение для $\varphi_{ij}^{(k)}$ и дополните процедуру Floyd-Warshall вычислением значений $\varphi_{ij}^{(k)}$.

Наконец, измените функцию Print-All-Pairs-ShortestPaths так, чтобы входом для неё служила матрица $\Phi = (\varphi_{ij}^{(n)})$. Объясните аналогию между матрицей Φ и таблицей s в задачи об оптимальной расстановке скобок в произведении матриц (раздел 16.1).

26.2-7 Придумайте алгоритм вычисления транзитивного замыкания ориентированного графа $G = (V, E)$ с временем работы $O(VE)$.

26.2-8 Предположим, что транзитивное замыкание ориентированного *ациклического* графа может быть построено за время $f(V, E)$, где $f(V, E) = \Omega(V + E)$ и f — монотонно возрастающая функция. Покажите, что тогда транзитивное замыкание произвольного ориентированного графа может быть построено за время $O(f(V, E))$.

Алгоритм Джонсона для разреженных графов

Алгоритм Джонсона находит кратчайшие пути для всех пар вершин за время $O(V^2 \lg V + VE)$, и поэтому для достаточно разреженных графов эффективнее повторного возведения в квадрат матрицы смежности графа и алгоритма Флойда-Уоршолла.

Алгоритм Джонсона либо возвращает матрицу весов кратчайших путей, либо сообщает, что в графе имеется цикл отрицательного веса. Этот алгоритм содержит вызовы описанных в главе 25 алгоритмов Дейкстры и Беллмана-Форда.

Алгоритм Джонсона основан на идее *изменения весов* (reweighting). Если веса всех рёбер графа неотрицательны, то можно найти кратчайшие пути между всеми парами вершин, применив алгоритм Дейкстры к каждой вершине. Используя фибоначчиевы кучи, мы можем сделать это за время $O(V^2 \lg V + VE)$. Если же в графе имеются ребра с отрицательным весом, то можно попытаться свести задачу к случаю неотрицательных весов, изменив весовую функцию. При этом должны выполняться такие свойства:

1. Кратчайшие пути не изменились: для любой пары вершин $u, v \in V$, кратчайший путь из u в v с точки зрения весовой функцией w

является также кратчайшим путём с точки зрения $\hat{w}$ и наоборот.

2. Все новые веса $\hat{w}(u, v)$ неотрицательны.

Как мы увидим, новая весовая функция $\hat{w}$ с такими свойствами может быть построена за время $O(VE)$.

Кратчайшие пути сохраняются

Следующая лемма указывает простой общий способ изменить весовую функцию, не меняя кратчайших путей. В её формулировке длины кратчайших путей с весовыми функциями w и $\hat{w}$ обозначаются через δ и $\hat{\delta}$ соответственно.

Лемма 26.1 (Изменение весов не меняет кратчайших путей)

Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w : E \to \mathbb{R}$. Пусть $h : V \to \mathbb{R}$ — произвольная функция с вещественными значениями, определённая на вершинах графа. Рассмотрим новую весовую функцию

$$\hat{w}(u, v) = w(u, v) + h(u) - h(v). \tag{26.9}$$

Тогда (a) произвольный путь $p = \langle v_0, v_1, \ldots, v_k \rangle$ является кратчайшим относительно w тогда и только тогда, когда он будетт кратчайшим относительно $\hat{w}$; (b) граф G содержит цикл с отрицательным w-весом тогда и только тогда, когда он содержит цикл с отрицательным $\hat{w}$-весом.

Доказательство. Оба утверждения леммы следуют из равенства

$$\hat{w}(p) = w(p) + h(v_0) - h(v_k). \tag{26.10}$$

В самом деле, из него следует, что для путей с фиксированными началом и концом разница между старым и новым весом постоянна (тем самым один и тот же путь будет кратчайшим). Кроме того, видно, что для циклов суммарный вес в старом и новом смысле одинаков, так как разница $h(v_0) - h(v_k)$ обращается в 0.

Равенство (26.10) проверить легко. Для каждого ребра пути имеем

$$\hat{w}(v_i, v_{i+1}) = w(v_i, v_{i+1}) + h(v_i) - h(v_{i+1}).$$

Если мы теперь сложим эти равенства для всех рёбер, то промежуточные члены сократятся, и получится равенство (26.10).

Как получить неотрицательные веса?

Теперь надо подобрать функцию h так, что бы изменённые веса $\hat{w}(u, v)$ были неотрицательны. Это можно сделать следующим образом. По данному ориентированному взвешенному графу $G = (V, E)$ с весовой функцией w построим новый граф $G' = (V', E')$ с одной дополнительной вершиной s (другими словами, $V' = V \cup \{s\}$), из которой идут рёбра нулевого веса во все вершины графа V ($E' = E \cup \{(s, v) : v \in V\}$ и $w(s, v) = 0$ для всех $v \in V$). Заметим, что вершина s может быть лишь начальной вершиной в пути (входящих в s рёбер нет). Очевидно, что новый граф G' содержит цикл отрицательного веса тогда и только тогда, такой цикл есть в исходном графе G.

На рис. 26.6 (a) показан граф G', соответствующий графу G рис. 26.1.

Предположим теперь, что граф G (а потому и G') не содержит цикла отрицательного веса. Положим $h(v) = \delta(s, v)$ для $\forall v \in V'$. Согласно лемме 25.3, для всех рёбер $(u, v) \in E'$ выполнено неравенство $h(v) \leqslant h(u) + w(u, v)$, которое можно переписать как $w(u, v) + h(u) - h(v) \geqslant 0$. Другими словами, новая весовая функция, определённая формулой (26.9), неотрицательна. На рис. 26.6(b) показаны новые веса рёбер графа G' рис. 26.6(a).

Вычисление кратчайших путей

Как мы видим, отыскание кратчайших путей для произвольного графа без циклов отрицательного веса можно сделать в два приёма: сначала мы ищем кратчайшие пути из одной вершины s (для чего годится алгоритм Беллмана–Форда), а после этого изменяем веса и получаем граф с неотрицательными весами, в котором ищем кратчайшие пути с помощью алгоритма Дейкстры, применённого последовательно ко всем вершинам.

Запишем соответствующую процедуру. Мы считаем, что граф задан с помощью списков смежных вершин. Процедура возвращает матрицу $D = (d_{ij})$ размера $|V| \times |V|$, где $d_{ij} = \delta_{ij}$, или же сообщает о наличии в графе циклов с отрицательными весами. (Мы предполагаем, что вершины графа G пронумерованы от 1 до $|V|$.)

```
{\sc Johnson}$(G)$\\
\verb|1 |создать граф $G'$, для которого $V[G']=V[G]\cup\{s\}$ и\\
\verb|     |$E[G']=E[G]\cup\{(s,v):\ v\in V[G]\}$\\
\verb|2 |if {\sc Bellmann-Ford}$(G',w,s)=${\sc false}\\
\verb|3    |then print ``имеется цикл отрицательного веса''\\
\verb|4    |else for (для) каждой вершины $v\in V[G']$\\
\verb|5          |do $h(v)\leftarrow\delta(s,v)$,\\
\verb|             |(значение $\delta(s,v)$ вычи-
слено алгоритмом
                            Беллмана--Форда)\\
\verb|6       |for (для) каждого ребра $(u,v)\in E[G']$\\
\verb|7          |do $\hat w(u,v)\leftarrow w(u,v)+h(u)-
h(v)$\\
\verb|8       |for (для) каждой вершины $u\in V[G]$\\
\verb|9          |do {\sc Dijkstra}$(G,\hat w,u)$\\
\verb|             |(вычисление $\hat\delta(u,v)$
                      для всех $v\in V[G])$\\
\verb|10         |for (для) каждой вершины $v\in V[G]$\\
\verb|11            |do $d_{uv}\leftarrow\hat\delta(u,v)+h(v)-
h(u)$\\
\verb|12 |return $D$
```

Рисунок 26.5 26.6 Алгоритма Джонсона в применении к графу рис. 26.1. (a) Граф G' с исходной весовой функций w. Вспомогательная вершина s — чёрная. Внутри каждой вершины v записано значение $h(v) = \delta(s, v)$. (b) Каждому ребру (u, v) приписан новый вес $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$. (c)–(g) Результаты работы алгоритма Дейкстры для всех вершин графа G с весовой функцитq $\hat{w}$. В каждом случае начальная вершина выделена чёрным. Внутри каждой вершины записаны величины $\hat{\delta}(u, v)/\delta(u, v)$. Вес кратчайшего пути $d_{uv} = \delta(u, v)$ равен $\hat{\delta}(u, v) + h(v) - h(u)$.

Эта процедура следует описанной схеме. В строке 1 формируется граф G', затем в строке 2 к нему применяется алгоритм Беллмана-Форда (при этом используется весовая функция w). Если в G' (а значит, и в G) есть цикл отрицательного веса, то об этом сообщается в строке 3. Строки 4–11 выполняются, если в графе такого цикла нет. В строках 4–5 вычисляются значения функции $h(v)$ для всех вершин $v \in V'$ (веса кратчайших путей $\delta(s, v)$, найденные алгоритмом Беллмана-Форда). В строках 6–7 вычисляются новые веса рёбер. В строках 8–11 для каждой вершины $u \in V[G]$ вызывается алгоритм Дейкстры и определяются веса кратчайших путей $\hat{\delta}(u, v)$ из этой вершины, а затем они пересчитываются в $\delta(u, v)$ по формуле (26.10), и ответ помещается в массиве D.

На рис. 26.6 показан пример выполнения алгоритма Джонсона.

Нетрудно подсчитать, что время работы алгоритма Джонсона составит $O(V^2 \lg V + VE)$, еслит использовать фибоначчиеву кучи для хранения очереди с приоритетами в в алгоритме Дейкстры. При более простой реализации очереди в виде двоичной кучи общее время работы составит $O(VE \lg V)$, что меньше, чем оценка $O(V^3)$ для алгоритма Флойда-Уоршолла, если только граф достаточно разрежен.

Упражнения

26.3-1 Примените алгоритм Джонсона к графу рис. 26.2, найдя значения h и $\hat{w}$.

26.3-2 Каким образом используется вспомогательная вершина в в алгоритме Джонсона?

26.3-3 Применим алгоритм Джонсона к графу, в котором исходная весовая функция сама неотрицательна. Что можно сказать о новой весовой функции в этом случае?

26.4* Замкнутые полукольца: общая схема для задач о путях

В этом разделе мы введём понятия замкнутого полукольца и дадим общую схему решения задач о путях в ориентированных графах. При этом алгоритм Флойда–Уоршолла и алгоритм построения транзитивного замыкания графа окажутся частными случаями этой общей схемы.

Определение замкнутого полукольца

Замкнутым полукольцом $(S, \oplus, \odot, \bar{0}, \bar{1})$ (по-английски closed semiring) называется множество S с определёнными на нем бинарными операциями *сложения* $\oplus$ и *умножения* $\odot$ (в английском оригинале используются термины summary и extension), а также элементами $\bar{0}, \bar{1} \in S$, для которого выполнены такие свойства:

1. $(S, \oplus, \bar{0})$ является *моноидом* (monoid):

- S *замкнуто* (is closed) относительно $\oplus$, то есть $a \oplus b \in S$ для всех $a, b \in S$;
- операция $\oplus$ ассоциативна (is associative): $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ для всех $a, b, c \in S$;
- $\bar{0}$ — нейтральный элемент (is an identity) для $\oplus$, то есть $a \oplus \bar{0} = \bar{0} \oplus a = a$ для всех $a \in S$;

$(S, \odot, \bar{1})$ также является моноидом;

2. Элемент $\bar{0}$ — аннулятор (annihilator): $a \odot \bar{0} = \bar{0} \odot a = \bar{0}$ для всех $a \in S$;

3. Операция $\oplus$ коммутативна (is commutative): $a \oplus b = b \oplus a$ для всех $a, b \in S$.

4. Операция $\oplus$ идемпотентна (is idempotent): $a \oplus a = a$;

5. Операция $\odot$ дистрибутивна относительно $\oplus$ (distributes over $\oplus$): $a \odot (b \oplus c) = (a \odot b) \oplus (a \odot c)$ и $(b \oplus c) \odot a = (b \odot a) \oplus (c \odot a)$ для всех a, b, c.

6. Можно продолжить операцию $\oplus$ на бесконечные последовательности, задач для любой последовательности $a_1, a_2, a_3, \ldots$ элементов S её сумму $a_1 \oplus a_2 \oplus a_3 \oplus \cdots \in S$. При этом нули (элементы $\bar{0}$) в бесконечной сумме можно выкидывать; если при этом она превращается конечную, то должна совпадать с суммой в прежнем смысле.

7. Операция бесконечного суммирования обладает свойствами ассоциативности, коммутативности и идемпотентности. (Тем самым, в любой бесконечной сумме можно произвольно менять порядок слагаемых и удалять повторяющиеся элементы, без изменения результата.)

8. Операция $\odot$ дистрибутивна относительно бесконечных сумм: $a \odot (b_1 \oplus b_2 \oplus b_3 \oplus \ldots) = (a \odot b_1) \oplus (a \odot b_2) \oplus (a \odot b_3) \oplus \ldots$ и $(a_1 \oplus a_2 \oplus a_3 \oplus \cdots) \odot b = (a_1 \odot b) \oplus (a_2 \odot b) \oplus (a_3 \odot b) \oplus \cdots$.

Исчисление путей в ориентированных графах

Замкнутые полукольца, несмотря на столь абстрактное определение, тесно связаны с путями в ориентированных графах. Рассмотрим ориентированный граф $G = (V, E)$ с заданной на нём *функцией разметки* (labeling function) $\lambda : V \times V \to S$. Мы будем считать, что эта функция задана не только на рёбрах, но и на любых парах вершин, полагая, что для пар (u, v), не являющихся рёбрами, значение $\lambda(u, v)$ равно $\bar{0}$, так что по существу функция задаётся метками на рёбрах (labels of edges)

Определим теперь понятие *метки пути* (path label). Именно, для любого пути $p = \langle v_1, v_2, \ldots, v_k \rangle$ мы будем называть его меткой $\lambda(p)$

произведение

$$\lambda(p) = \lambda(v_1, v_2) \odot \lambda(v_2, v_3) \odot \cdots \odot \lambda(v_{k-1}, v_k).$$

Единичный элемент $\bar{1}$ для умножения будет меткой пустого пути (длины 0). Если путь p не является путём в графе (то есть при некотором i пара (v_i, v_{i+1}) не является ребром), то в выражении для $\lambda(p)$ один из множителей и потому всё произведение (см. пункт 2 определения замкнутого полукольца) равны $\bar{0}$.

Мы будем иллюстрировать применение замкнутых полуколец на примере задачи о кратчайших путях (для случая неотрицательных весов). В качестве S возьмём множество $S = \mathbb{R}^{\geqslant 0} \cup \{\infty\}$ (множество неотрицательных вещественных чисел с добавленной (плюс) бесконечностью). Тогда веса всех рёбер будут элементами S. Операцией *"умножения" $\odot$ будет обычное сложение; чтобы не запутаться, отныне мы будем брать в кавычки слова "произведение", "сумма" и т.п., если они относятся к операиям в полукольце. Тогда " произведением" меток на рёбрах вдоль пути будет сумма весов рёбер, то есть вес пути. " Единицей" полукольца будет нейтральный элемент относительно сложения, то есть* 0*: можно написать* $\bar{1} = 0$. *Метка пустого пути (обозначим его ε) будет равна нулю:* $\lambda(\varepsilon) = w(\varepsilon) = 0 = \bar{1}$.

Возвращаясь к общей ситуации, определим соединение, *или* конкатенацию *(concatenation) двух путей* $p_1 = \langle v_1, v_2, \ldots, v_k \rangle$ *и* $p_2 = \langle v_k, v_{k+1}, \ldots, v_l \rangle$ *как путь*

$$p_1 \circ p_2 = \langle v_1, v_2, \ldots, v_k, v_{k+1}, \ldots, v_l \rangle,$$

(соединение имеет смысл, если конец первого пути совпадает с началом второго). Ассоциативность "умножения" гарантирует нам, что метка соединения путей равна "произведению" их меток:

$$\begin{array}{rcl} \lambda(p_1 \circ p_2) & = & \lambda(v_1, v_2) \odot \lambda(v_2, v_3) \odot \cdots \odot \lambda(v_{k-1}, v_k) \odot \\ & & \lambda(v_k, v_{k+1}) \odot \lambda(v_{k+1}, v_{k+2}) \odot \cdots \odot \lambda(v_{l-1}, v_l) \\ & = & (\lambda(v_1, v_2) \odot \lambda(v_2, v_3) \odot \cdots \odot \lambda(v_{k-1}, v_k)) \odot \\ & & (\lambda(v_k, v_{k+1}) \odot \lambda(v_{k+1}, v_{k+2}) \odot \cdots \odot \lambda(v_{l-1}, v_l)) \\ & = & \lambda(p_1) \odot \lambda(p_2) \end{array} .$$

Как мы увидим, различные задачи о путях могут рассматриваться как частные случаи такой общей задачи. Дано замкнутое полукольцо и граф с метками на рёрах; найти (для всех пар вершин $i, j \in V$) суммы меток по всем возможным путям из i в j:

$$l_{ij} = \bigotimes_{i \overset{p}{\leadsto} j} \lambda(p). \tag{26.11}$$

Эта "сумма" может быть бесконечной (путей из i в j может быть бесконечно много). При её вычислении можно включать и

Рисунок 26.6 26.7 Сумму меток путей $p_1 \circ p_2$ и $p_1 \circ p_3$ можно записать как $(\lambda(p_1) \odot \lambda(p_2)) \oplus (\lambda(p_1) \odot \lambda(p_3))$. По свойству дистрибутивности это выражение равно $\lambda(p_1) \odot (\lambda(p_2) \oplus \lambda(p_3))$.

Рисунок 26.7 26.8 Благодаря циклу c имеется бесконечно много путей из вершины v в вершину x, а именно, $p_1 \circ p_2$, $p_1 \circ c \circ p_2$, $p_1 \circ c \circ c \circ p_2$ и т.д.

пути, проходящие по отсутствующим в графе рёбрам: как мы договорились, метки таких рёбер равны $\bar{0}$, *поэтому и метки этих путей равны* 0 *и по нашему предположению (свойство 6) на "сумму" такие пути не влияют. Коммутативность и ассоциативность оператора* $\oplus$ *(свойство 7) позволяют нам не указывать порядок путей при суммировании.*

Вернёмся к нашему примеру, в котором элементами S *были неотрицательные действительные числа и символ* ∞, *а "умножением" было обычное сложение. Если мы теперь определим "сложение" как взятие минимума (точнее, точной нижней грани, так как мы должны определить его и для бесконечных последовательностей), то получим замкнутое полукольцо (свойства 1–8 легко проверить). Отметим, что элемент* $\bar{0}{=}\infty$ *является нейтральным элементом для "сложения":* $\min(a, \infty) = a$.

Если считать меткой ребра его вес, то меткой пути будет также его вес (сумма весов рёбер), а уравнение (26.11) определяет l_{ij} *как точную нижнюю грань весов всех путей из* i *в* j.

Понятие полукольца позволяет выполнять алгебраические преобразования с метками путей. Пример такого рода приведён на рис. 26.7.

В более сложных случаях число путей может быть бесконечно. Такой пример приведён на рис. 26.8. Здесь (считаем, что других путей из u *в* x *нет) формула (26.11) приводит к "сумме"*

$$\lambda(p_1) \odot \lambda(p_2) \oplus \lambda(p_1) \odot \lambda(c) \odot \lambda(p_2) \oplus \lambda(p_1) \odot \lambda(c) \odot \lambda(c) \odot \lambda(p_2) \oplus \ldots =$$

$$= \lambda(p_1) \odot \left(\bar{1} \oplus \lambda(c) \oplus \lambda(c) \odot \lambda(c) \oplus \ldots\right) \odot \lambda(p_2)$$

Для краткой записи такой "суммы" введём следующее определение. Пусть a *— произвольный элемент замкнутого полукольца* S. Замыканием *(closure) элемента* a *назовём выражение*

$$a^* = \bar{1} \oplus a \oplus (a \cdot a) \oplus (a \cdot a \cdot a) \oplus \ldots$$

Тогда предыдущее выражение (для рис. 26.8) можно записать как $\lambda(p_1) \cdot (\lambda(c))^* \cdot \lambda(p_2)$.

В нашем примере полукольца (соответствующего задаче о кратчайших путях) имеем $a^* = \min\{ka | k = 0, 1, 2, \ldots\} = 0$ *для любого элемента* $a \geqslant 0$. *Что и не удивительно: если цикл имеет неотрицательный вес, то идти по нему нет смысла.*

Примеры замкнутых полуколец

Один такой пример мы уже расмотрели: полукольцо $S_1 = \{\mathbb{R}^{\geqslant 0} \cup \{\infty\}, \min, +, \infty, 0\}$*. Мы можем расширить это полукольцо, разрешив отрицательные элементы, а также элемент* $-\infty$*. Получится полукольцо* $S_2 = \{\mathbb{R} \cup \{+\infty\} \cup \{-\infty\}, \min, +, +\infty, 0\}$ *(упр. 26.4-3). Это полукольцо можно использовать при доказательстве правильности алгоритма Флойда–Уоршалла для случая отриццательных весов. Отметим, что теперь*

$$a^* = \begin{cases} 0, \text{если } a \geqslant 0, \\ -\infty, \text{если } a < 0. \end{cases}$$

Второй случай ($a < 0$) *говорит нам, что цикл отрицательного веса можно проходить многократно, и веса будут стремиться к* $-\infty$.

Задаче о транзитивном замыкании соответствует замкнутое полукольцо $S_3 = (\{0,1\}, \vee, \wedge, 0, 1)$*. При этом все рёбра исходного графа имеют пометку* 1 *(а отсутствующим соответствует значение* $\bar{0} = 0$*, как мы говорили). В этом полукольце значение* l_{ij}*, вычисленное по формуле (26.11), равно* 1*, если пара* (i,j) *принадлежит транзитивному замыканию (то есть есть путь из* i *в* j*), и равно* 0 *в противном случае. Отметим, что для этого кольца* $1^* = 0^* = 1$.

Динамическое программирование и сумма меток по путям

Покажем, как можно вычислить выражение (26.11) с помощью алгоритма, аналогичного алгоритму Флойда–Уоршалла и алгоритму вычисления транзитивного замыкания.

Напомним, что нам дано замкнутое полукольцо S *и ориентированный граф* (G,V)*, рёбра которого помечены элементами* S*. Мы хотим для каждой пары вершин* i,j *вычислить " сумму" (в смысле полукольца)*

$$l_{ij} = \bigoplus \lambda(p)$$

где "суммирование" происходит по всем путям из i *в* j*, а* $\lambda(p)$ *есть метка пути* p*, то есть "произведение" меток на его рёбрах.*

Рассмотрим величину $l_{ij}^{(k)}$*, которая получится, если ограничить суммирование только теми путями, в которых все промежуточные вершины лежат в множестве* $\{1,2,\ldots,k\}$*. Для* $l_{ij}^{(k)}$ *можно написать рекурентное соотношение:*

$$l_{ij}^{(k)} = l_{ij}^{(k-1)} \oplus (l_{ik}^{(k-1)} \odot (l_{kk}^{(k-1)})^* \odot l_{kj}^{(k-1)}). \qquad (26.12)$$

Это формула напоминает рекурентные соотношения (26.5) и (26.8); разница в том, что в ней есть "множитель" $(l_{kk}^{(k-1)})^*$*, соответствующий "сумме" меток всех циклов, начинающихся и* *кончающихся в* k. *(Почему не было такого множителя в алгоритме Флойда–Уоршалла с неотрицательными весами и в алгоритме вычисления транзитивного замыкания? Дело в том, что в обоих случаях* $a^* = \bar{1}$ *и поэтому этот " множитель" можно было опустить.)*

Начальные значения для рекуррентного соотношения (26.12) таковы:

$$l_{ij}^{(0)} = \begin{cases} \lambda(i,j) & \text{если } i \neq j, \\ \bar{1} \oplus \lambda(i,j) & \text{если } i = j. \end{cases}$$

В самом деле, путь из одного ребра имеет метку, равную метку этого ребра, а пустой путь имеет метку $\bar{1}$ *в соответствии с нашим соглашением; этот путь надо учесть при* $i = j$.

Мы приходим к алгоритму, использующему метод динамического программирования для последовательного отыскания величин $l_{ij}^{(k)}$ *при* $k = 0, 1, 2 \ldots, n$; *его ответом является матрица* $L^{(n)} = (l_{ij}^{(n)})$ *(соответствующая " суммированию" по всем путям без ограничений на промежуточные вершины).*

```
{\sc Compute-Summaries}$(\lambda,V)$\\
\verb|1 |$n\leftarrow|V|$\\
\verb|2 |for $i\leftarrow1$ to $n$\\
\verb|3     |do for $j\leftarrow1$ to $n$\\
\verb|4             |do if $i=j$\\
\verb|5                   |then $l^{(0)}_{ij}\leftarrow\bar1\oplus\lambda(i,
\verb|6                   |else $l^{(0)}_{ij}\leftarrow\lambda(i,j)$\\
\verb|7 |for $k\leftarrow1$ to $n$\\
\verb|8     |do for $i\leftarrow1$ to $n$\\
\verb|9             |do for $j\leftarrow1$ to $n$\\
\verb|10                   |do $l^{(k)}_{ij}=l^{(k-1)}_{ij}\oplus(l^{(k-
1)}_{ik}\
odot(l^{(k-1)}_{kk})^{\ast}\odot l^{(k-1)}_{kj})$\\
\verb|11 |return $L^{(n)}$
```

Время работы данного алгоритма зависит от времени выполнения операций $\odot$, $\oplus$ *и* *. *Обозначив время выполнения операций через* $T_\odot$, $T_\oplus$, T_*, *общее время работы алгоритма* Compute-Summaries *можно записать как* $\Theta(n^3)(T_\odot + T_\oplus + T_*))$, *что превращается в* $\Theta(n^3)$, *если время выполнения любой из трёх операций составляет* $O(1)$.

Упражнения

26.4-1 Проверьте, что $S_1 = (\mathbb{R}^{\geqslant 0} \cup \{\infty\}, \min, +, \infty, 0)$ *и* $S_3 = (\{0,1\}, \vee, \wedge, 0, 1\})$ *являются замкнутыми полукольцами.*

26.4-2 Проверьте, что $S_2 = (\mathbb{R} \cup \{-\infty, +\infty\}, \min, +, +\infty, 0)$ *является закнутым полукольцом. Чему равно значение* $a + (-\infty)$ *для* $a \in \mathbb{R}$? *Что можно сказать о значении* $(-\infty) + (+\infty)$?

26.4-3 Запишите алгоритм Compute-Summaries *для случая замкнутого полукольца S_2, аналогичный алгоритму Флойда-Уоршолла. Чему должно быть равно значение $-\infty + \infty$, чтобы алгоритм правильно искад длины кратчайших путей?*

26.4-4 Является ли $S_4 = (\mathbb{R}, +, \cdot, 0, 1)$ замкнутым полукольцом? (Точнее следовало бы спросить так: можно ли его превратить в замкнутое полукольцо, определив каким-либо образом сумму любого бесконечного ряда?)

26.4-5 Можем ли мы обобщить на случай произвольного замкнутого полукольца алгоритм Дейкстры? Алгоритм Беллмана-Форда? Процедуру Faster-All-Pairs-Shortest-Paths*?*

26.4-6

Мы хотим узнать, какой наиболее тяжёлый грузовик может проехать из Горюнина в Простоквашино, имея карту дорог между этими сёлами, в которой для каждой дороги указан максимально возможный вес грузовика. Постройте алгоритм для решения этой задачи, использующий подходящее замкнутое полукольцо.

Задачи

26-1 Транзитивное замыкание растущего графа

Мы хотим вычислять транзитивное замыкание ориентированного графа $G = (V, E)$, множество рёбер которого растёт. Другими словами, мы хотим обновлять транзитивное замыкание после добавления в граф ещё одного ребра. Мы считаем, что изначально граф G не имел рёбер вовсе. Транзитивное замыкание должно храниться в булевой матрице.

a. Покажите, как за время $O(V^2)$ можно произвести обновление транзитивного замыкания после добавления в граф G нового ребра.

b. Покажите, что любой алгоритм обновления транзитивного замыкания (хранящий его в булевой матрице) может потребовать времени $O(V^2)$ после добавления одного ребра.

c. Постройте алгоритм обновления транзитивного замыкания графа после добавления рёбер, для которого суммарное время работы при добавлении любой последовательности рёбер не превосходит $O(V^3)$.

26-2 Кратчайшие пути в ε-плотных графах

Граф $G = (V, E)$ называется ε-плотным, *если $|E| = \Theta(V^{1+\varepsilon})$ для некоторой константы ε, лежащей в диапазоне $0 < \varepsilon \leqslant 1$. Использование d-ичных куч (см. задачу 7-2) в алгоритме поиска кратчайших путей на ε-плотных графах позволяет избавиться от использования фибоначчиевых куч (довольно сложной структуры данных), сохранив ту же асимптотическую оценку времени работы.*

a. Определите асимптотику времени работы процедур Insert, Extract-Min *и* Decrease-Key *как функцию от d и n (здесь n —*

число элементов d-ичной кучи). Что получается при $d = \Theta(n^{\alpha})$, где $0 < \alpha \leqslant 1$ — некоторая константа. Сравните эти времена с учётными временами этих операций для фибонначиевых куч.

b. Придумайте способ вычислить кратчайшие пути из одной вершины в ε-плотном ориентированном графе $G = (V, E)$ без рёбер отрицательного веса за время $O(E)$. (Указание: выберите подходящее d как функцию ε.)

c. Покажите, что можно вычислить кратчайшие пути для всех пар вершин в ε-плотном ориентированном графе $G = (V, E)$ без рёбер отрицительного веса за время $O(VE)$.

d. Покажите, что за время $O(VE)$ можно вычислить кратчайшие пути для всех пар вершин в ε-плотном ориентированном графе $G = (V, E)$, который может иметь рёбра отрицательного веса, но не содержит циклов отрицательного веса.

26-3 Минимальный остов и замкнутое полукольцо

Пусть $G = (V, E)$ — связный неориентированный граф с весовой функций $w : E \to \mathbb{R}$, вершинами которого являются числа от 1 до n. Предположим, что веса $w(i, j)$ всех рёбер различны. Пусть T — единственный (см. упр. 24.1-6) минимальный остов графа G. Магс (B.M.Maggs) и С.А.Плоткин предложили использовать замкнутое полукольцо для отыскания минимального покрывающего дерева. Для каждой пары вершин определим минимаксный *вес (minimax weight) m_{ij} взяв минимум по всем путям максимумов весов рёбер на каждом пути.*

a. Покажите, что $S = (\mathbb{R} \cup \{-\infty, \infty\}, \min, \max, \infty, -\infty)$ является замкнутым полукольцом.

Таким образом, мы можем использовать процедуру Compute-Summaries *для вычисления минимаксных весов m_{ij} в графе G. Обозначим через $m_{ij}^{(k)}$ минимаксный вес для всех путей из i в j с промежуточными вершинами из множества $\{1, 2, \cdots, k\}$.*

b. Напишите рекуррентную формулу для $m_{ij}^{(k)}$ при $k \geqslant 0$.

c. Пусть $T_m = \{(i, j) \in E : w(i, j) = m_{ij}\}$. Докажите, что ребра из T_m образуют покрывающее дерево для графа G.

d. Покажите, что это покрывающее дерево будет минимальным. (Указание. Посмотрите, что происходит при добавлении ребра (i, j) в T и одновременном удалении из T какого-нибудь ребра, лежащего на пути из i в j. Посмотрите также, что происходит при замене ребра (i, j) из T на другое ребро.)

Замечания

Лоулер [132] подробно рассматривает задачу нахождения кратчайших путей для всех пар вершин, хотя и не выделяет отдельно случай разреженных графов (при этом связь задачи о кратчайших путях с умножением матриц относится к фольклору). Алгоритм Флойда–Уоршолла описан в работе Флойда [68], который опирается результат Уоршолла [198] (о транзитивным замыка-

нии булевых матриц). Понятие замкнутого полукольца появилось в книге Ахо, Хопкофта и Ульмана [4]. Алгоритм Джонсона взят из [114].

27 Максимальный поток

Рассмотрим ориентированный граф. Будем рассматривать его как сеть труб, по которым некоторое вещество движется от истока (где оно производится с некоторой постоянной скоростью) к стоку (где оно потребляется — с той же скоростью). Вместо потоков вещества можно рассматривать движение тока по проводам, деталей по конвейеру, информации по линиям связи или товаров от производителя к потребителю.

Как и в задаче о кратчайших путях, на каждом ребре графа мы пишем число. Но если там это число означало длину пути, то теперь это скорее ширина дороги, или пропускная способность трубы — максимальная скорость потока в этой трубе. Например, она может быть 200 литров в час, или 20 ампер (если речь идёт об электричестве).

Мы считаем, что в вершинах вещество не накапливается — сколько приходит, столько и уходит (если вершина не является истоком или стоком). Это свойство называется "законом сохранения потока" (flow conservation). Для электрического тока это свойство называется первым правилом Кирхгофа.

Задача о максимальном потоке для данной сети состоит в следующем: найти максимально возможную скорость производства (и потребления) вещества, при которой его ещё можно доставить от истока к стоку при данных пропускных способностях труб. В этой главе после точной формулировки этой задачи (раздел 27.1) описаны два метода её решения. В разделе 27.2 разобран классический метод Форда–Фалкерсона. Его использование для поиск максимального паросочетания в двудольном неориентированном графе описано в разделе 27.3. Раздел 27.4 излагает схему " проталкивания предпотока" которая используется во многих современных алгоритмах для решения задач о потоках в сетях. Один из таких алгоритмов описан в разделе 27.5. Хотя он и не самый быстрый из известных (время работы $O(V^3)$), но он использует те же идеи, что и самые быстрые алгоритмы, и достаточно эффективен на практике.

27.1 Потоки в сетях

В этом разделе мы дадим точное определение сетей и потоков в них, обсудим их свойства, сформулируем задачу о максимальном потоке и введём некоторые полезные обозначения.

Сети и потоки

Назовем сетью *(flow network) ориентированный граф* $G = (V, E)$, *каждому ребру* $(u, v) \in E$ *которого поставлено в соответствие число* $c(u, v) \geqslant 0$, *называемое* пропускной способностью *(capacity) ребра. В случае* $(u, v) \notin E$ *мы полагаем* $c(u, v) = 0$. *В графе выделены две вершины:* исток *(source)* s *и* сток *(sink)* t. *Для удобства мы предполагаем, что в графе нет "бесполезных" вершин (каждая вершина* $v \in V$ *лежит на каком-то пути* $s \rightsquigarrow v \rightsquigarrow t$ *из истока в сток). (В таком случае граф связен и* $|E| \geqslant |V| - 1$.*) Пример сети показан на рис.27.1.*

Теперь дадим определение потока. Пусть дана сеть $G = (V, E)$, *пропускная способность которой задаётся функцией* c. *Сеть имеет исток* s *и сток* t. Потоком *(flow) в сети* G *назовём функцию* $f : V \times V \to R$, *удовлетворяющую трём свойствам:*

Ограничение, связанное с пропускной способностью *(capacity constraint):* $f(u, v) \leqslant c(u, v)$. *для всех* u, v *из* V.

Кососимметричность *(skew symmetry):* $f(u, v) = -f(v, u)$ *для всех* u, v *из* V.

Сохранение потока *(flow conservation):*

$$\sum_{v \in V} f(u, v) = 0.$$

для всех u *из* $V - \{s, t\}$.

Величина $f(u, v)$ *может быть как положительной, так и отрицательной. Она определяет, сколько вещества движется из вершины* u *в вершину* v *(отрицательные значения соответствуют движению в обратную сторону).*

Величина *(value) потока* f *определяется как сумма*

$$\sum_{v \in V} f(s, v). \tag{27.1}$$

(складываем потоки по всем рёбрам, выходящим из истока). Она обозначается $|f|$ *(не спутайте с абсолютной величиной числа!).* Задача о максимальном потоке *(maximum-flow problem) состоит в следующем: для данной сети* G *с истоком* s *и стоком* t *найти поток максимальной величины.*

Поясним смысл трёх наших свойств. Первое означает, что поток из одной вершины в другую не превышает пропускной способности ребра. Второе представляет собой соглашение о том,

Рисунок 27.1 27.1. (a) Сеть $G = (V, E)$, описывающие возможности перевозок продукции фирмы *"Кленовые листья"*. Исток s — фабрика в Ванкувере, сток t — склад в Виннипеге. На каждом ребре написано максимальное число ящиков, которые можно отправить в день. (b) Пример потока f в сети G величины 19. Показаны только положительные значения $f(u, v) > 0$ (после косой черты стоит пропускная способность $c(u, v)$.

что отрицательные числа соответствуют потоку в обратную сторону. Из него следует также, что $f(u, u) = 0$ для любой вершины u (положим $u = v$). Третье свойство означает, что для любой вершины u (кроме стока и истока) сумма потоков во все другие вершины равна нулю. Учитывая кососимметричность, это свойство можно переписать как

$$\sum_{u \in V} f(u, v) = 0$$

(теперь переменная суммирования обозначена u) и прочесть так: "сумма всех потоков из других вершин равна нулю".

Заметим также, что если вершины u и v не соединены ребром, то поток между ними, то есть $f(u, v)$, равен нулю. Действительно, если $(u, v) \notin E$ и $(v, u) \notin E$, то $c(u, v) = c(v, u) = 0$. Тогда из первого свойства следует, что $f(u, v) \leqslant 0$ и $f(v, u) \leqslant 0$. Вспоминая, что $f(u, v) = -f(v, u)$ (кососимметричность), мы видим, что $f(u, v) = f(v, u) = 0$.

Разделим вещество, поступающее в данную вершину v и вещество, из неё выходящее (то есть положительные и отрицательные значения $f(u, v)$). Сумму

$$\sum_{\substack{u \in V \\ f(u,v) > 0}} f(u, v). \tag{27.2}$$

назовём входящим (в вершину v) потоком. Выходящий поток определяется симметрично. Теперь закон сохранения потока можно сформулировать так: для любой вершины, кроме истока и стока, входящий поток равен исходящему.

Пример сети

Рассмотрим пример на рис. 27.1 (a). Компания "Кленовые листья" производит хоккейные шайбы на фабрике в Ванкувере (исток s) и складирует их в Виннипеге (сток t). Она арендует место в грузовиках другой фирмы, и место это ограничено: из города u в город v можно доставить не более $c(u, v)$ ящиков в день. Ограничения $c(u, v)$ показаны на рисунке. Задача состоит в том, чтобы перевозить максимально возможное количество шайб из Ванкувера в Виннипег ежедневно. При этом путь может занимать несколько дней, и ящики могут ждать отправки в промежуточных пунктах, но необходимо, чтобы для каждого пункта

Рисунок 27.2 27.2 Сокращение. (a) Вершины v_1 и v_2. Здесь $c(v_1, v_2) = 10$ и $c(v_2, v_1) = 4$. (b) Ежедневно 8 ящиков перевозят из v_1 в v_2. (c) Добавили встречные перевозки 3 ящиков в день из v_2 в v_1. (d) Сократили противоположные потоки — осталось 5 ящиков в день. (e) Добавили перевозку ещё 7 ящиков в день из v_2 в v_1.

число ежедневно прибывающих ящиков было равно числу увозимых (иначе ящиков не хватит или они будут накапливаться). Тем самым выполнено свойство сохранения потока. Величиной потока будет число шайб, ежедневно отгружаемых из Ванкувера, и нас интересует поток максимальной величины.

Один из возможных потоков показан на рис.27.1(b). Из вершины u в вершину v в день отправляется $f(u, v)$ ящиков; если $f(u, v)$ равно 0, ящики не отправляются; отрицательные значения $f(u, v)$ соответствуют ящикам, прибывающим в u из v.

Внимательный читатель мог уже заметить, что реальная ситуация не вполне описывается нашей моделью. Именно, наша модель не учитывает встречные перевозки. Если из вершины v_1 в v_2 ежедневно везут восемь ящиков, а из v_2 в v_1 ежедневно везут три ящика, чему должны быть равны $f(v_1, v_2)$ и $f(v_2, v_1)$? (Напомним, что эти величины должны быть противоположны.) Мы полагаем $f(v_1, v_2) = 8 - 3 = 5$, а $f(v_2, v_1) = -5$. Те же значения функции f соответствуют ежедневным перевозкам пяти ящиков из v_1 в v_2, так что в нашей модели встречные перевозки автоматически сокращаются. Впрочем, и в реальности встречные перевозки разумно " сократить" друг с другом, (экономика должна быть экономной), при этом места на грузовиках понадобится только меньше.

На рис. 27.2 показан пример сокращения потоков между вершинами v_1 и v_2 (27.2 (a)). Сначала из v_1 в v_2 возили ежедневно 8 ящиков (27.2 (b)). Затем стали возить 3 ящика в обратном направлении (27.2 (c)), пока не догадались вместо этого уменьшить число перевозимых в обратную сторону ящиков на 3 (27.2 (d)). Эти две различные (в жизни) ситуации соответствуют одной и той же функции f: в обоих случаях $f(v_1, v_2) = 5$, а $f(v_2, v_1) = -5$. Если теперь руководство требует начать перевозки дополнительных 7 ящиков в день из v_2 в v_1, то нужно прежде всего отменить перевозки 5 ящиков в обратную сторону, после чего назначить перевозку дополнительных 2 ящиков (27.2 (e)). Тем самым требование будет выполнено (несмотря на то, кстати, что в грузовиках из v_2 в v_1 есть места только на 4 ящика.

Сети с несколькими истоками и стоками

Можно рассматривать задачу о максимальном потоке для случая нескольких истоков и стоков. Компания "Кленовые листья" может иметь m фабрик $\{s_1, s_2, \ldots s_m\}$ и n складов $\{t_1, t_2, \ldots, t_n\}$ (рис. 27.3 (a)). Но это не усложняет дела, потому что такой

Рисунок 27.3 27.3 Задача о максимальном потока для нескольких истоков и стоков сводится к обычной. (a) Сеть с пятью истоками $S = \{s_1, s_2, s_3, s_4, s_5\}$ и тремя стоками $T = \{t_1, t_2, t_3\}$. (b) Сооответствующая сеть с одним истоком и одним стоком; добавленные рёбра имеют бесконечную пропускную способность.

вариант проблемы можно свести к обычному. На рис. 27.3 (b) показана эквивалентная сеть с одним истоком и одним стоком. Мы добавили общий исток *(supersource) s, из которого ведут рёбра бесконечной пропускной способности во все прежние истоки ($c(s, s_i) = \infty$ при всех $i = 1, 2, \ldots, m$). Аналогичным образом из всех прежних стоков проведены рёбра в* общий сток *(supersink) t. Легко видеть, что каждый поток в сети (a) соответствует потоку в сети (b) и наоборот (формальное доказательство оставляется читателю в качестве упр. 27.1-3)*

Обозначения

*Мы будем использовать следующее соглашение: если в выражении на месте вершины стоит множество вершин, то имеется в виду сумма по всем элементам этого множества (*неявное суммирование, *implicit summary notation). Это относится и к случаю нескольких переменных. Например, если X и Y — множества вершин, то*

$$f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y).$$

В этих обозначениях закон сохранения потока запишется как $f(u, V) = 0$ для всех $u \in V - \{s, t\}$. Кроме того, в неявных суммах мы опускаем фигурные скобки (например, в равенстве $f(s, V \setminus s) = f(s, V)$ символ $V \setminus s$ обозначает $V \setminus \{s\}$.

Вот несколько полезных свойств таких сумм (доказательство мы оставляем читателю как упр. 27.1-4):

Лемма 27.1

Пусть f — поток в сети $G = (V, E)$. Тогда для любого $X \subseteq V$ выполнено

$$f(X, X) = 0.$$

Для любых $X, Y \subseteq V$ выполнено

$$f(X, Y) = -f(Y, X).$$

Для любых $X, Y, Z \subseteq V$ из $X \cap Y = \emptyset$ следует

$$f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$$

и

$$f(Z, X \cup Y) = f(Z, X) + f(Z, Y).$$

Для примера докажем с использованием таких обозначений, что величина потока равна сумме потоков из всех вершин в сток:

$$|f| = f(V, t). \tag{27.3}$$

Интуитивно это ясно (куда ж ему ещё деваться), но можно провести и формальное рассуждение. Вот оно:

По определению,

$$|f| = f(s, V)$$

Применяя лемму 27.1, имеем

$$f(s, V) = f(V, V) - f(V \setminus s, V) = f(V, V \setminus s) = f(V, t) + f(V, V \setminus s \setminus t).$$

По закону сохранения потока второе слагаемое равно 0, *и остаётся* $f(V, t)$.

Обобщение леммы 27.1 будет доказано ниже (лемма 27.5)

Упражнения

27.1-1

Следуя образцу рис. 27.2, изобразите две вершины u *и* v, *для которых* $c(u, v) = 5$, $c(v, u) = 8$, *из* u *в* v *пересылаются* 3 *единицы, а из* v *в* u — 4. *Чему равен поток из* u *в* v*?*

27.1-2

Проверьте, что функция f *рис.27.1 (b) действительно является потоком.*

27.1-3

Как приспособить определение потока для случая нескольких истоков и стоков? Покажите, что задача о максимальном потоке для такого случая сводится к обычной с помощью описанного нами приёма.

27.1-4

Докажите лемму 27.1.

27.1-5

Для сети $G = (V, E)$ *и потока* f *на рис.27.1 (b), укажите пример подмножеств* $X, Y \subseteq V$, *для которых* $f(X, Y) = -f(V \setminus X, Y$, *а также подмножеств* $X, Y \subseteq V$, *для которых* $f(X, Y) \neq -f(V \setminus X, Y$.

27.1-6

Пусть имеется сеть $G = (V, E)$ *и два потока* f_1 *и* f_2 *на ней. Рассмотрим их сумму (функцию из* $V \times V$ *в* $\mathbb{R}$*):*

$$(f_1 + f_2)(u, v) = f_1(u, v) + f_2(u, v) \tag{27.4}$$

Каким требованиям из определения потока она удовлетворяет обязательно, а каким может не удовлетворять?

27.1-7

Поток f *можно умножить на вещественное число* α, *получив функцию*

$$(\alpha f)(u, v) = \alpha \cdot f(u, v)$$

Докажите, что для любой сети множество потоков выпукло. Это значит, что если f_1 *и* f_2 — *потоки, то сумма* $\alpha f_1 + (1 - \alpha) f_2$ *при* $0 \leqslant \alpha \leqslant 1$ *тоже является потоком.*

27.1-8

Сформулируйте задачу о максимальном потоке как задачу линейного программирования.

27.1-9

Рассмотрим сеть с несколькими веществами *(multicommodity flow network) в которой имеются потоки p веществ, для каждого из которых есть свой исток и свой сток. Для каждого вещества есть свой закон сохранения потока, а пропускная способность одна на всех (сумма потоков всех веществ из u в v не должна превышать $c(u,v)$). Для каждого вещества имеется своя величина потока; складывая эти величины, получим* суммарную величину *(total flow value) потока. Докажите, что поток максимальной суммарной величины можно найти за полиномиальное время, представив эту задачу как задачу линейного программирования. (Любая задача линейного программирования разрешима за полиномиальное время: этот факт мы упоминали в разделе 25.5.) упп даст по три числа, бо́льших x, за двумя возможными исключениями: группа, содержащая x, и последняя неполная группа. Тем самым имеется не менее*

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil \right\rceil - 2\right) \geqslant \frac{3n}{10} - 6.$$

элементов, заведомо больших x, и точно так же получаем, что имеется не менее $3n/10 - 6$ элементов, заведомо меньших x. Значит, алгоритм Select*, рекурсивно вызываемый на пятом шаге, будет обрабатывать массив длиной не более $7n/10 + 6$.*

Пусть теперь $T(n)$ — время работы алгоритма Select *на массиве из n элементов в худшем случае. Первый, второй и четвертый шаги выполняются за время $O(n)$ (на втором шаге мы $O(n)$ раз сортируем массивы размером $O(1)$), третий шаг выполняется за время не более $T(\lceil n/5\rceil)$, а пятый шаг, по доказанному, — за время, не превосходящее $T(\lfloor 7n/10{+}6\rfloor)$ (как и раньше, можно предполагать, что $T(n)$ монотонно возрастает с ростом n). Стало быть,*

$$T(n) \leqslant T(\lceil n/5\rceil) + T(\lfloor 7n/10 + 6\rfloor) + O(n).$$

Поскольку сумма коэффициентов при n в правой части ($1/5 + 7/10 = 9/10$) меньше единицы, из этого рекуррентного соотношения вытекает, что $T(n) \leqslant cn$ для некоторой константы c. Это можно доказать по индукции. В самом деле, предполагая, что $T(m) \leqslant cm$ для всех $m < n$, имеем

$$\begin{aligned}
T(n) &\leqslant c(\lceil n/5\rceil) + c(\lfloor 7n/10 + 6\rfloor) + O(n) \\
&\leqslant c(n/5 + 1) + c(7n/10 + 6) + O(n) \\
&\leqslant 9cn/10 + 7c + O(n) = \\
&= cn - c(n/10 - 7) + O(n).
\end{aligned}$$

При подходящем выборе c это выражение будет не больше cn при всех $n > 70$ (надо, чтобы $c(n/10 - 6)$ превосходило коэффициент, подразумеваемый в $O(n)$). Таким образом, индуктивный переход возможен при $n > 70$ (заметим ещё, что при таких n выражения $\lceil n/5 \rceil$ и $\lfloor 7n/10 + 6 \rfloor$ меньше n).

Увеличив c ещё (если надо), можно добиться того, чтобы $T(n)$ не превосходило cn и при всех $n \leqslant 70$, что завершает рассуждение по индукции. Стало быть, алгоритм Select *работает за линейное время (в худшем случае).*

Отметим, что алгоритмы Select *и* Randomized-Select*, в отличие от описанных в главе 9 алгоритмов сортировки за линейное время, используют только попарные сравнения элементов массива и применимы для произвольного упорядоченного множества. Эти алгоритмы асимптотически эффективнее очевидного подхода "упорядочи множество и выбери нужный элемент", поскольку всякий алгоритм сортировки, использующий только попарные сравнения, требует времени $\Omega(n \lg n)$ не только в худшем случае (раздел 9.1), но и в среднем (задача 9-1).*

Упражнения

27.1-1 *Будет ли алгоритм* Select *работать за линейное время, если разбивать массив на группы не из пяти, а из семи элементов? Покажите, что для групп из трёх элементов рассуждение не проходит.*

27.1-2 *Пусть x — "медиана медиан" в алгоритме* Select *(массив содержит n элементов). Покажите, что при $n \geqslant 38$ количество элементов, бо́льших x (так же как и количество элементов, меньших x) не меньше $\lceil n/4 \rceil$.*

27.1-3 *Модифицируйте алгоритм быстрой сортировки так, чтобы он работал за время $O(n \lg n)$ в худшем случае.*

27.1-4* *Пусть алгоритм выбора i-го по счёту элемента использует только попарные сравнения. Покажите, что с помощью тех же сравнений можно в качестве побочного результата получить списки элементов, меньших искомого, а также бо́льших искомого.*

27.1-5 *Пусть у нас есть какой-то алгоритм, находящий медиану за линейное в худшем случае время. Используя его в качестве подпрограммы, разработайте простой алгоритм, решающий задачу нахождения произвольной порядковой статистики за линейное время.*

Рисунок 27.4 Как провести с востока на запад магистраль, чтобы суммарная длина подводящих трубопроводов была минимальна?

27.1-6 *Под* k-**квантилями** *(k-th quantiles) множества из* n *чисел мы понимаем* $k-1$ *его элементов, обладающих следующим свойством: если расположить элементы множества в порядке возрастания, то квантили будут разбивать множество на* k *равных (точнее, отличающихся не более чем на один элемент) частей. Разработайте алгоритм, который за время* $O(n \lg k)$ *находит* k*-квантили данного множества.*

27.1-7 *Разработайте алгоритм, который по заданному* k *находит в данном множестве* S *его* k *элементов, менее всего отстоящих от медианы. Число операций должно быть* $O(|S|)$.

27.1-8 *Пусть* $X[1\,..\,n]$ *и* $Y[1\,..\,n]$ *— два возрастающих массива. Разработайте алгоритм, находящий за время* $O(\lg n)$ *медиану множества, полученного объединением элементов этих массивов.*

27.1-9 *Профессор консультирует нефтяную компанию, которой требуется провести магистральный нефтепровод в направлении строго с запада на восток через нефтеносное поле, на котором расположены* n *нефтяных скважин. От каждой скважины необходимо подвести к магистрали трубопровод по кратчайшему пути (строго на север или на юг, рис. 10.2). Координаты всех скважин профессору известны; необходимо выбрать местоположение магистрали, чтобы сумма длин всех трубопроводов, ведущих от скважин к магистрали, была минимальна. Покажите, что оптимальное место для магистрали можно найти за линейное время.*

Задачи

27-1 Сортировка i наибольших элементов

Дано множество из n чисел; требуется выбрать из них i наибольших и отсортировать (пользуясь только попарными сравнениями). Для каждого из приведенных ниже подходов разработайте соответствующий алгоритм и выясните, как зависит от n и i время работы этих алгоритмов в худшем случае.

а *Отсортировать все числа и выписать i наибольших.*

б *Поместить числа в очередь с приоритетами и вызвать i раз процедуру* EXTRACT-MAX.

в *Найти с помощью алгоритма раздела 10.3 i-е по величине число (считая от наибольшего), разбить массив относительно него и отсортировать i наибольших чисел.*

27-2 Взвешенная медиана

Пусть дано n различных чисел $x_1, \ldots, x_n$, и пусть каждому x_i сопоставлено положительное число ("вес") w_i, причём сумма всех весов равна 1. **Взвешенной медианой** *(weighted median) называется такое число x_k, что*

$$\sum_{x_i < x_k} w_i \leqslant \frac{1}{2} \quad и \quad \sum_{x_i > x_k} w_i \leqslant \frac{1}{2}.$$

а *Покажите, что если все веса равны $1/n$, то взвешенная медиана совпадает с обычной.*

б *Как найти взвешенную медиану n чисел с помощью сортировки за время $O(n \lg n)$ в худшем случае?*

в *Как модифицировать алгоритм* SELECT *(раздел 10.3), чтобы он искал взвешенную медиану за время $\Theta(n)$ в худшем случае?*

Задача о выборе места для почты *(post-office location problem) состоит в следующем. Дано n точек $p_1, \ldots, p_n$ и n положительных весов $w_1, \ldots, w_n$; требуется найти точку p (не обязательно совпадающую с одной из p_i), для которой выражение $\sum_{i=1}^{n} w_i d(p, p_i)$ будет минимально (через $d(a, b)$ обозначается расстояние между точками a и b).*

д *Покажите, что в одномерном случае (точки — вещественные числа, $d(a, b) = |a - b|$) взвешенная медиана будет решением этой задачи.*

е *Найдите оптимальное решение в двумерном случае (точки — пары вещественных чисел), если расстояние между точками $a = (x_1, y_1)$ и $b = (x_2, y_2)$ задается "в L_1-метрике": $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$ (американцы называют такую метрику Manhattan* *distance, по названию района Нью-Йорка, разбитого улицами на прямоугольные кварталы)*

27-3 Нахождение i-го по величине элемента при малых i

Пусть $T(n)$ — время работы процедуры Select*, примененной к массиву из n чисел; в худшем случае $T(n) = \Theta(n)$, но коэффициент при n, подразумеваемый в этом обозначении, довольно велик. Если i мало по сравнению с n, то отобрать i-ый по величине элемент можно быстрее.*

а *Опишите алгоритм, который находит i-й по величине элемент в множестве из n чисел, делая $U_i(n)$ сравнений, причём*

$$U_i(n) = \begin{cases} T(n) & \text{если } n \leqslant 2i, \\ n/2 + U_i(\lfloor n/2 \rfloor) + T(2i+1) & \text{иначе.} \end{cases}$$

(Указание: сделайте $\lfloor n/2 \rfloor$ попарных сравнений и отберите i наименьших среди меньших элементов пар.)

б *Покажите, что $U_i(n) = n + O(T(2i)\log(n/i))$.*

в *Покажите, что, при постоянном i, имеем $U_i(n) = n + O(\lg n)$. [Та же оценка получится, если построить кучу из главы 7, а затем i раз выбрать из неё минимальный элемент.]*

г *Пусть $i = n/k$, причём $k \geqslant 2$; покажите, что $U_i(n) = n + O(T(2n/k)\lg k)$.*

Замечания

Алгоритм для нахождения медианы за линейное в худшем случае время предложили Блюм, Флойд, Пратт, Ривест и Тарьян [29]. Вероятностный алгоритм с линейным средним временем работы принадлежит Хоару [97]. Флойд и Ривест [70] разработали усовершенствованную версию этого алгоритма, в которой граница разбиения определяется по небольшой случайной выборке.

27.2 Метод Форда–Фалкерсона

В этом разделе мы рассмотрим метод Форда–Фалкерсона отыскания максимального потока. Мы говорим о методе, а не об алгоритме, поскольку есть несколько алгоритмов, его реализующих и отличающихся временем работы. Одна из таких реализаций приведена в конце раздела. Ключевую роль в методе Фода–Фалкерсона играют три понятия: остаточные сети, дополняющие пути и разрезы. Основная теорема — теорема 27.7 о максимальном потоке и минимальном разрезе.

Поиск максимального потока методом Форда–Фалкерсона проводится последовательно. Вначале поток нулевой (и величина его равна нулю). На каждом шаге мы увеличиваем значение потока. Для этого мы находим "дополняющий путь", по которому можно пропустить ещё немного вещества, и используем его для увеличения потока. Этот шаг повторяется, пока есть дополняющие пути. Как покажет теорема о максимальном потоке и минимальном разрезе, полученный поток будет максимальным. Запишем этот план с помощью псевдокода:

```
Ford-Fulkerson-Method($G,s,t$)

положить поток $f$ равным $0$
while (пока) существует дополняющий путь $p$
 do  дополнить $f$ вдоль $p$
return $f$
```

Остаточные сети

Пусть дана сеть и поток в ней. Неформально говоря, остаточная сеть состоит из тех рёбер, поток по которым можно увеличить. Строгое определение таково: пусть $G = (V, E)$ — сеть с истоком s и стоком t. Пусть f — поток в этой сети. Для любой пары вершин u и v. рассмотрим остаточную пропускную способность *(residual capacity) из u в v, определяемую как*

$$c_f(u, v) = c(u, v) - f(u, v). \tag{27.5}$$

Она определяет, сколько ещё потока можно направить из u в v. Например, если $c(u, v) = 16$, а $f(u, v) = 11$ то мы можем переслать ещё $c_f(u, v) = 5$ единиц по ребру (u, v). Остаточная пропускная способность $c_f(u, v)$ может превосходить $c(u, v)$, если в данный момент поток $f(u, v)$ отрицателен. Например, если $c(u, v) = 16$, а $f(u, v) = -4$, то $c_f(u, v) = 20$. В самом деле, мы можем увеличить поток на 4, отменив встречный поток, и ещё отправить 16 единиц, не превышая пропускной способности ребра (u, v).

Сеть $G_f = (V, E_f$, где

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$$

назовём остаточной сетью *(residual network) сети G, порождённой потоком f. Её рёбра, называемые* остаточными рёбрами *(residual edges) допускают положительный поток. На рис. 27.4 (a) изображен поток f в сети G (взятый с рис. 27.1 (b)), а на рис. 27.4 (b) изображена остаточная сеть G_f.*

Заметим, что остаточное ребро (u, v) не обязано быть ребром сети G. Иными словами, может оказаться, что $E_f \nsubseteq E$. Рёбер

(v_1, s) и $v_2, v_3)$ на рис. 27.4 (b) не было в исходной сети. Такое ребро из u в v появляется, когда $f(u,v) < 0$, то есть когда имеется имеется поток вещества в обратном направлении (по ребру (v,u)) — ведь этот поток можно уменьшить. Таким образом, если ребро (u,v) принадлежит остаточной сети, то хотя бы одно из рёбер (u,v) и (v,u) было в исходной сети. Получаем оценку

$$|E_f| \leqslant 2|E|.$$

Остаточная сеть G_f является сетью с пропускными способностями c_f. Следующая лемма показывает, как соотносятся потоки в исходной и в остаточной сетях.

Лемма 27.2

Пусть $G = (V,E)$ — сеть с истоком s и стоком t, а f — поток в ней. Пусть G_f — остаточная сеть сети G, порождённая потоком f. Пусть f' — поток в G_f. Тогда сумма $f + f'$, определенная как в (27.4), является потоком в сети G величины $|f + f'| = |f| + |f'|$.

Доказательство

Сначала докажем, что $f + f'$ будет потоком. Проверим кососимметричность. Для всех $u, v \in V$ выполнено

$$\begin{aligned}(f+f')(u,v) &= f(u,v) + f'(u,v) = \\ &= -f(v,u) - f'(v,u) = \\ &= -(f(v,u) + f'(v,u)) = \\ &= -(f+f')(v,u).\end{aligned}$$

Проверим условие, связанное с ограниченной пропускной способностью. Заметим, что $f'(u,v) \leqslant c_f(u,v)$ для всех $u, v \in V$, поэтому

$$(f+f')(u,v) = f(u,v)+f'(u,v) \leqslant f(u,v)+(c(u,v)-f(u,v)) = c(u,v).$$

Проверим закон сохранения потока. Для всех $u \in V \setminus \{s,t\}$ выполнено равенство $(f + f')(u,V) = f(u,V) + f'(u,V) = f(u,V) + f'(u,V) = 0 + 0 = 0$.

Наконец, найдём величину суммарного потока: $|f + f'| = (f + f')(s,V) = f(s,V) + f'(s,V) = |f| + |f'|$.

Дополняющие пути

Пусть f — поток в сети $G = (V,E)$. Назовём дополняющим путём *(augmenting path) простой путь из истока s в сток t в остаточной сети G_f. Из определения остаточной сети вытекает, что по всем ребрам (u,v) дополняющего пути можно переслать ещё сколько-то вещества, не превысив пропускную способность ребра.*

Рисунок 27.5 27.4 (a) Поток f в сети G (как на рис. 27.1). (b) Остаточная сеть G_f. Выделен дополняющий путь p. Его остаточная пропускная способность $c_f(p)$ равна $c(v_2, v_3) = 4$. (c) Результат добавления потока величины 4, проходящего вдоль пути p. (d) Остаточная сеть, порождённая потоком рис. (c).

На рис. 27.4 (b) дополняющий путь выделен серым цветом. По нему можно отправить еще 4 *единицы потока, так как наименьшая остаточная пропускная способность рёбер этого пути равна* $c(v_2, v_3) = 4$. *Величину наибольшего потока, который можно переслать по дополняющему пути* p, *назовём* остаточной пропускной способностью *(residual capacity) пути* p:

$$c_f(p) = \min\{c_f(u, v) : (u, v) \in p\}$$

Сказанное уточняется в следующей лемме (доказательство мы оставляем читателю в качестве упр. 27.2-3).

Лемма 27.3

Пусть f *— поток в сети* $G = (V, E)$ *и* p *— дополняющий путь в* G_f. *Определим функцию* $f_p : V \times V \to \mathbb{R}$ *так:*

$$f_p(u, v) = \begin{cases} c_f(p), \text{ если } (u, v) \in p \\ -c_f(p), \text{ если } (v, u) \in p \\ 0 \text{ в остальных случаях.} \end{cases} \qquad (27.6)$$

Тогда f_p *— поток в сети* G_f *и* $|f_p| = c_f(p) > 0$.

Теперь видно, что если добавить поток f_p *к потоку* f, *получится поток в сети* G *с бóльшим значением. На рис. 27.4 (c) изображён результат добавления потока* f_p *(рис. 27.4 (b)) к потоку* f *(рис. 27.4 (a)). Сформулируем это ещё раз:*

Следствие 27.4

Пусть f *— поток в сети* $G = (V, E)$, *а* p *— дополняющий путь в сети* G_f, *заданный равенством (27.6). Тогда функция* $f' = f + f_p$ *является потоком в сети* G *величины* $|f'| = |f| + |f_p| > |f|$.

Доказательство

Утверждение вытекает из лемм 27.2 и 27.3.

Разрезы в сетях

Метод Форда–Фалкерсона добавляет последовательно потоки по дополняющим путям, пока не получится максимальный поток. Как мы вскоре увидим (теорема о максимальном потоке и минимальном разрезе), величина потока максимальна в том и только в том случае, когда остаточная сеть не содержит дополняющих путей. Для доказательства нам понадобится понятие разреза сети.

Назовём разрезом *(cut) сети* $G = (V, E)$ *разбиение множества* V *на две части* S *и* $T = V \setminus S$, *для которых* $s \in S$ *и* $t \in T$. *(Подобная процедура делалась для минимального покрывающего дерева в главе 24, но теперь граф ориентирован и, кроме того, мы*

Рисунок 27.6 27.5 Разрез (S,T) в сети рис. 27.1 (b). Здесь $S = \{s, v_1, v_2\}$ (черные вершины) и $T = \{v_3, v_4, t\}$ (белые вершины). При этом $f(S,T) = 19$ (поток через разрез) и $c(S,T) = 26$ (пропускная способность)

требуем, чтобы $s \in S$, $t \in T$.) Пропускной способностью разреза *(capacity of the cut) (S,T) называют сумму $c(S,T)$. Кроме того, для заданного потока f величина* потока через разрез (S,T) *определяется как сумма $f(S,T)$.*

На рис. 27.5 изображён разрез $(\{s, v_1, v_2\}, \{v_3, v_4, t\})$ сети рис. 27.1 (b). Поток через этот разрез равен

$$f(v_1, v_2) + f(v_2, v_3) + f(v_2, v, 4) = 12 + (-4) + 11 = 19,$$

а пропускная способность разреза равна

$$c(v_1, v_3) + c(v_2, v_4) = 12 + 14 = 26.$$

Как видно, поток через разрез, в отличие от пропускной способности разреза, может включать и отрицательные слагаемые.

Следующая лемма утверждает, что величины потоков через все разрезы одинаковы (и равны величине потока).

Лемма 27.5

Пусть f — поток в сети G с истоком s и стоком t, а (S,T) — разрез сети G. Тогда поток через разрез (S,T) равен $f(S,T) = |f|$.

Доказательство Многократно используя лемму 27.1, получаем

$$\begin{aligned} f(S,T) &= f(S,V) - f(S,S) = \\ &= f(S,V) = \\ &= f(s,V) + f(S \setminus s, V) = \\ &= f(s,V) = \\ &= |f| \end{aligned}$$

Доказанное выше равенство (27.3) (величина потока равна потоку в сток) немедленно следует из этой леммы.

Следствие 27.6

Значение любого потока f в сети G меньше или равно пропускной способности любого разреза сети G.

Доказательство

Пусть (S,T) — произвольный разрез сети G. В силу 27.5 и ограничений на потоки по рёбрам

$$\begin{aligned} |f| &= f(S,T) = \\ &= \sum_{u \in S} \sum_{v \in T} f(u,v) \leqslant \\ &\leqslant \sum_{u \in S} \sum_{v \in T} c(u,v) = \\ &= c(S,T). \end{aligned}$$

Теперь докажем основную теорему этого раздела (max-flow min-cut theorem).

Теорема 27.7 (о максимальном потоке и минимальном разрезе)

Пусть f — поток в сети $G = (V, E)$. Тогда следующие утверждения равносильны:

1. Поток f максимален (является потоком максимальной величины) в сети G.

2. Остаточная сеть G_f не содержит дополняющих путей.

3. Для некоторого разреза (S, T) сети G выполнено равенство $|f| = c(S, T)$. (В этом случае, как показывает следствие 27.6, разрез является минимальным, то есть имеет минимально возможную пропускную способность.)

Доказательство

$(1) \Rightarrow (2)$

Рассуждая от противного, допустим, что поток f максимален, но G_f содержит дополняющий путь p. Рассмотрим сумму $f + f_p$, где f_p задается равенством (27.6). По следствию 27.4 эта сумма является потоком в G, величина которого больше $|f|$, что противоречит максимальности f.

$(2) \Rightarrow (3)$

Пусть в сети G_f нет пути из истока s в сток t. Рассмотрим множество

$$S = \{v \in V \mid \text{в } G_f \text{ существует путь из } s \text{ в } v\}.$$

Положим $T = V \setminus S$. Очевидно, что $s \in S$, а $t \in T$, так как в G_f нет пути из s в t. Поэтому пара (S, T) — разрез. Ни для каких $u \in S$ и $v \in T$ ребро (u, v) не принадлежит E_f (в противном случае вершина v попала бы в S). Поэтому $f(u, v) = c(u, v)$. По лемме 27.5 $|f| = f(S, T) = c(S, T)$.

$(3) \Rightarrow (1)$

Для любого разреза (S, T) выполнено $|f| \leqslant c(S, T)$ (следствие 27.6). Поэтому из равенства $|f| = c(S, T)$ следует, что поток f максимален.

Общая схема алгоритма Форда–Фалкерсона

Действуя по методу Форда–Фалкерсона, на каждом шаге мы вибираем произвольный дополняющий путь p и увеличиваем поток f, добавляя поток величины $c_f(p)$ по пути p. Приводимый ниже алгоритм использует массив $f[u, v]$ для хранения текущих значения потока. Мы считаем, что функция $c(u, v)$ вычисляется за время $O(1)$, при этом $c(u, v) = 0$ если $(u, v) \notin E$. (При естественной реализации значение (u, v) хранится рядом с рёбрами в списках исходящих рёбер.)

В строке 5 величина $c_f(u, v)$ понимается в соответствии с формулой (27.5). Символ $c_f(p)$ обозначает локальную переменную, в которую помещается остаточная пропускная способность пути p.

Рисунок 27.7 27.6 Работа процедуры Ford-Fulkerson. (**a**)–(**d**) Шаги выполнения цикла. Слева изображена остаточная сеть G_f (дополняющий путь p выделен серым), справа показан увеличенный поток. Вначале (a) роль остаточной сети играет исходная сеть G. (**e**) Остаточная сеть более не содержит дополняющего пути; поток максимален.

```
Ford-Fulkerson($G,s,t$)

1 for (для) каждого ребра $(u,v)$ из $E[G]$
2   do $f[u,v]\leftarrow  0$
3      $f[v,u]\leftarrow  0$
4 while (пока) в остаточной сети $G_f$ существует путь $p$ из $s$ в $t$
5   do $c_f(p)\leftarrow\min\{c_f(u,v)|(u,v) \textrm{ вхо-
дит в } p\}$
6      for (для) каждого ребра $(u,v)$ пути $p$
7          do $f[u,v]\leftarrow f[u,v]+c_f(p)$
8             $f[v,u]\leftarrow -f[u,v]$
```

Процедура Ford-Fulkerson *следует описанной выше схеме (*Ford-Fulkerson-Method*). Строки 1–3 задают первоначальное значение потока; цикл в строках 4–8 на каждом шаге находит дополняющий путь p в G_f и увеличивает поток f. Если дополняющего пути нет, найденный поток максимален. Пример работы программы изображен на рис.27.6.*

Анализ алгоритма Форда–Фалкерсона

Время работы процедуры Ford-Fulkerson *зависит от того, как ищется путь p (строка 4). В принципе алгоритм может вообще не остановиться, если значение потока будет расти всё более мелкими шагами, так и не достигнув максимума. Однако если выбирать дополняющий путь при помощи поиска в ширину (раздел 23.2), то алгоритм работает полиномиальное время. Прежде чем доказать это, мы установим простую верхнюю оценку для случая целых пропускных способностей. (Этот случай часто встречается на практике. Случай рациональных пропускных способностей сводится к нему умножением на число.)*

Для указанного случая (пропускные способности — целые числа), процедура Ford-Fulkerson *выполняется за время $O(E|f^*|)$, где f^* — максимальный поток. В самом деле, выполнение строк 1-3 требует времени $\Theta(E)$. Цикл в строках 4–8 выполняется не более $|f^*|$ раз, так как после каждого выполнения величина потока увеличивается по крайней мере на единицу. Осталось оценить время одного шага, которое зависит от того, как мы храним данные о потоке. Рассмотрим ориентированный граф $G' = (V, E')$, в*

Рисунок 27.8 27.7 (**a**) Сеть, для которой процедура Ford-Fulkerson требует времени $\Theta(E|f^*|)$, где f^* — максимальный поток величины $|f^*| = 2\,000\,000$. Выделен дополняющий путь с пропускной способностью 1. (**b**) Полученная остаточная сеть. Выделен дополняющий путь с той же пропускной способностью. (**c**) Полученная остаточная сеть.

котором

$$E' = \{(u,v)|(u,v) \in E \text{ или } (v,u) \in E\},$$

и будем хранить потоки и пропускные способности рядом с соответствующими рёбрами (рёбра сети G входят и в граф G', так что пропускные способности есть где хранить). Остаточная сеть (для текущего потока) состоит из тех рёбер (u,v) графа G', для которых $c(u,v) - f[u,v] \neq 0$. Поиск дополняющего пути в остаточной сети (в глубину или в ширину — пока это всё равно) займет время $O(E) = O(E')$. Поэтому время работы процедуры будет $O(E|f^*|)$.

Из доказанной оценки следует, что при небольшом значении $|f^*|$ и целых пропускных способностях время работы процедуры Ford-Fulkerson *невелико. Но при большом $|f^*|$ время работы алгоритма может быть велико даже для простой сети, как показывает пример рис. 27.7. Значение максимального потока в этой сети равно* 2, 000, 000 *(если использовать рёбра, идущие слева направо). Если, как показано на рис. 27.7 (a) и 27.7 (b), на чётных шагах будет выбираться дополняющий путь $s \to v \to u \to t$, а на нёчетных — $s \to u \to v \to t$, то потребуется* 2, 000, 000 *шагов, чтобы найти максимальный поток.*

Можно получить лучшую оценку времени работы процедуры Ford-Fulkerson, *если предположить, что в строке 4 используется поиск в ширину. В этом случае путь p будет кратчайшим из дополняющих путей (длину каждого ребра считаем равной единице). Эта реализация метода Форда–Фалкерсона называется* алгоритмом Эдмондса–Карпа. *Докажем, что время работы алгоритма Эдмондса-Карпа равно $O(VE^2)$.*

Обозначим длину кратчайшего пути в G_f между вершинами u и v через $\delta_f(u,v)$).

Лемма 27.8

Рассмотрим работу алгоритма Эдмондса–Карпа на сети $G = (V,E)$ с истоком s и стоком t. Для любой вершины v из $V \setminus \{s,t\}$ расстояние $\delta_f(s,v)$ в остаточной сети между истоком и вершиной $v \in V \setminus \{s,t\}$ монотонно неубывает на каждом шаге цикла.

Доказательство

Предположим противное: пусть после увеличения потока f (вдоль дополняющего пути) расстояние $\delta_f(s,v)$ от истока s до некоторой вершины v из $V \setminus \{s,t\}$ уменьшилось. Обозначив увели-

ченный поток через f', получаем, что

$$\delta'_f(s,v) < \delta_f(s,v).$$

Среди вершин v с таким свойством выберем ближайшую (в смысле сети G'_f) к истоку. В этом случае

$$\text{из } \delta'_f(s,u) < \delta'_f(s,v) \text{ следует } \delta_f(s,u) \leqslant \delta'_f(s,u) \tag{27.7}$$

для всех $u \in V \setminus \{s,t\}$.

Возьмем кратчайший путь p' из s в v в сети G'_f и рассмотрим вершину u, непосредственно предшествующую вершине v в этом пути ($s \rightsquigarrow u \to v$).

Согласно следствию 25.2, $\delta'_f(s,u) = \delta'_f(s,v) - 1$. Следовательно, по предположению (27.7)

$$\delta_f(s,u) \leqslant \delta'_f(s,u).$$

Может ли ребро (u,v) входить в E_f (это значит, что $f[u,v] < c(u,v)$). Тогда по лемме 25.3 должно быть

$$\begin{aligned}\delta_f(s,v) &\leqslant \delta_f(s,u) + 1 \leqslant \\ &\leqslant \delta'_f(s,u) + 1 = \\ &= \delta'_f(s,v),\end{aligned}$$

что противоречит начальному предположению.

Следовательно, $(u,v) \notin E_f$. Но ребро (u,v) принадлежит E'_f, поэтому дополняющий путь p, превративший сеть G_f в сеть G'_f, содержал ребро (v,u). Так как путь p был кратчайшим путём из s в t в сети G_f, то любой его подпуть также был кратчайшим (лемма 25.1), поэтому $\delta_f(s,u) = \delta_f(s,v)+1$. Отсюда следует, что

$$\begin{aligned}\delta_f(s,v) &= \delta_f(s,u) - 1 \leqslant \\ &\leqslant \delta'_f(s,u) - 1 = \\ &= \delta'_f(s,v) - 2 < \\ &< \delta'_f(s,v),\end{aligned}$$

что противоречит начальному предположению.

Теорема 27.9

Алгоритм Эдмондса–Карпа на сети $G = (V,E)$ выполняется за $O(VE)$ шагов.

Доказательство

В каждом дополняющем пути отметим рёбра минимальной пропускной способности, которые будем называть критическими *(critical) (для данного шага). Заметим, что критические рёбра исчезают при переходе к следующей остаточной сети.*

Покажем, что для одно и то же ребро может быть критическим не более $O(V)$ раз. Поскольку все рёбра всех остаточных сетей соответствуют рёбрам графа (возможно, в обратном направлении), их число не превосходит $2E$, так что общее число шагов действительно есть $O(EV)$.

Итак, посмотрим, сколько раз за время работы алгоритма Эдмондса–Карпа ребро (u,v) может быть критическим. Если ребро было критическим, оно исчезает, и может появиться вновь лишь после того, как поток из u в v уменьшится. Это может произойти только если ребро (v,u) войдёт в дополняющий путь. Таким образом, возникает последовательность ситуаций:

(u,v) входит в дополняющий путь (и критическое в нём);

(v,u) входит в дополняющий путь (не обязательно критическое);

(u,v) входит в дополняющий путь (и критическое в нём);

(v,u) входит в дополняющий путь (не обязательно критическое);

и так далее. Покажем, что число таких ситуаций есть $O(V)$. Для этого проследим, как меняются расстояния от истока до вершин u и v в остаточных сетях. Как мы знаем (лемма 27.8), эти две величины могут только возрастать. При этом то одна, то другая вырывается вперёд на 1 (поскольку они соседствуют в кратчайшем пути, лемма 25.1). Заметим, что при переходе от первой ситуации ко второй расстояние от истока до u увеличится по крайней мере на 2 (оно было меньше расстояния до v, а стало больше). При переходе от второй строки к третьей расстояние от истока до v увеличится по крайней мере на 2 и так далее.

Заметим, что расстояния ограничены сверху значением V (кратчайший путь не проходит дважды через одну вершину), так что общее число таких чередований есть $O(V)$, как мы и обещали.

Подсчитаем теперь общее время работы алгоритма Эдмондса–Карпа. Каждая итерация требует времени $O(E)$, их будет $O(VE)$, поэтому всего будет $O(VE^2)$.

В разделе 27.4 мы расскажем о другом подходе к поиску максимального потока, который позволяет сделать это за $O(V^2E)$. Его усовершенствование позволяет ещё уменьшить время работы, получив оценку $O(V^3)$ (раздел 27.5).

Упражнения

27.2-1

Найдите поток через разрез $(\{s,v_2,v_4\},\{v_1,v_3,t\})$ на рис. 27.1 b). Чему равна пропускная способность этого разреза?

27.2-2

Проследите за выполнением лгоритма Эдмондса–Карпа для *сети рис. 27.1 (а).*

27.2-3

Укажите минимальный разрез на рис. 27.6, соответствующий нарисованному максимальному потоку. В каких случаях дополняющий путь уменьшал поток, идущий в обратном направлении?

27.2-4

Докажите, что для любых вершин u *и* v*, любого потока* f *и пропускной способности* c *выполнено* $c_f(u,v) + c_f(v,u) = c(u,v) + c(v,u)$.

27.2-5

Вспомним конструкцию из раздела 27.1, с помощью которой сеть с несколькими истоками и стоками сводится к обычной. Докажите, что если пропускная способность любого ребра исходной сети была конечной, то в получившейся сети величина любого потока конечна.

27.2-6

Рассмотрим сеть с несколькими истоками и стоками, в которой каждый исток s_i *производит ровно* p_i *единиц потока, а каждый сток* t_j *потребляет ровно* q_j *единиц, т.е.* $f(s_i, V) = p_i$ *и* $f(V, t_j) = q_j$. *При этом* $\sum_i p_i = \sum_j q_j$. *Как узнать, возможен ли поток с такими ограничениями, сведя эту задачу к задаче о максимальном потоке?*

27.2-7

Докажите лемму 27.3.

27.2-8

Покажите, что при поиске максимального потока в сети $G = (V, E)$ *достаточно сделать* $|E|$ *шагов, если только правильно выбрать дополняющие пути на каждом шаге. (Указание. Считая, что максимальный поток известен, покажите, как следует выбирать пути.)*

27.2-9

Назовём запасом связности *(edge connectivity) неориентированного графа минимальное число рёбер, которое необходимо удалить, чтобы сделать граф несвязным. Например, связность дерева равна единице, а связность цикла равна 2.*

Как узнать связность графа с помощью программы поиска максимального потока? Её следует применять не более чем к $|V|$ *сетям, каждая из которых содержит* $O(V)$ *вершин и* $O(E)$ *рёбер.*

27.2-10

Предположим, что для каждого ребра (u,v) *сети обратное ему ребро* (v,u) *также входит в сеть Покажите, что алгоритм Эдмондса–Карпа требует не более* $|V| \cdot |E|/4$ *итераций. (Указание: для каждого ребра* (u,v) *проследите, как меняются* $\delta(s,u)$ *и* $\delta(v,t)$ *между теми моментами, когда ребро* (u,v) *критическое.)*

Рисунок 27.9 27.8 Двудольный граф. Множество вершин разбито на две части L и R. (**a**) Паросочетание из двух рёбер. (**b**) Максимальное паросочетание состоит из трёх элементов.

27.3 Максимальное паросочетание в двудольном графе

Некоторые комбинаторные задачи (например, задача о сети с несколькими истоками и стоками из раздела 27.1) сводятся к разобранной нами задаче о максимальном потоке. Другой пример такого рода — задача о максимальном паросочетании в двудольном графе (см. раздел 5.4). В этом разделе покажем, как с помощью метода Форда-Фалкерсона можно решить эту задачу для графа $G = (V, E)$ *за время* $O(VE)$.

Задача о максимальном паросочетании

Пусть $G = (V, E)$ *— неориентированный граф.* Паросочетанием *(matching) назовем множество рёбер* $M \subseteq E$*, не имеющих общих концов (каждая вершина* $v \in V$ *является концом максимум одного ребра из* M*). Будем говорить, что вершина* $v \in V$ входит *в паросочетание* M *(is matched), если в* M *есть ребро с концом* v*; в противном случае* v свободна *(is unmatched).* Максимальное паросочетание *(maximum matching) — это паросочетание* M*, содержащее максимально возможное число рёбер (*$|M| \geqslant |M'|$ *для любого паросочетания* M'*). Пример паросочетания приведён на рис. 27.8.*

В этом разделе мы будем рассматривать паросочетания лишь в двудольных графах. Мы предполагаем, что множество V *разбито на два непересекающихся подмножества* L *и* R*, и любое ребро из* E *соединяет некоторую вершину из* L *с некоторой вершиной из* R.

Для задачи о максимальном паросочетании в двудольном графе есть несколько метафор. Вот наиболее известная: L *— женихи,* R *— невесты, наличие ребра* (u, v) *означает, что* u *и* v *согласны стать супругами. Максимальное паросочетание доставляет ЗАГСу больше всего работы.*

Поиск максимального паросочетания в двудольном графе

Мы будем использовать метод Форда–Фалкерсона для поиска максимального паросочетания в двудольном графе $G = (V, E)$ *за полиномиальное от* $|V|$ *и* $|E|$ *время. Для этого рассмотрим сеть* $G' = (V', E')$*, соответствующую двудольному графу* G *(рис. 27.9). Эта сеть строится так: добавляются две новые вершины, которые будут истоком (s) и стоком (t):* $V' = V \cup \{s, t\}$*. Множе-*

Рисунок 27.10 27.9 Сеть, соответствующая двудольному графу. (**a**) Двудольный граф рис. 27.8. Выделенные рёбра образуют максимальное паросочетание. (**b**) Соответствующая сеть G' и максимальный поток в ней. Пропускная способность любого ребра равна единице; поток по выделенным рёбрам равен единице, по остальным — нулю. Выделенные рёбра, соединяющие вершины из L с вершинами из R, соответствуют максимальному паросочетанию в двудольном графе.

ство (направленных) рёбер сети G' таково:

$$\begin{aligned} E' =&\{(s,u)|u\in L\}\cup \\ &\cup\{(u,v)|u\in L, v\in R, (u,v)\in E\}\cup \\ &\{(v,t)|v\in R\}. \end{aligned}$$

(напомним, что через L и R обозначаются доли графа). Будем считать, что пропускная способность каждого ребра равна единице.

Следующая лемма устанавливает соответствие между потоками в G' и паросочетаниями в G — но прежде дадим ещё одно определение.

Поток f в сети $G = (V, E)$ называется целочисленным *(integer-valued), если все значения $f(u, v)$ — целые.*

Лемма 27.10

Пусть $G = (V, E)$ — двудольный граф с долями L и R, и $G' = (V', E')$ — соответствующая сеть. Пусть M — паросочетание в G. Тогда существует целочисленный поток в G' со значением $|f| = |M|$. Обратно, если f — целочисленный поток в G', то в G' найдется паросочетание из $|f|$ элементов.

Доказательство

Сначала докажем, что паросочетание порождает поток, задав поток f следующим образом. Если $(u, v) \in M$, то $f(s, u) = f(u, v) = f(v, t) = 1$ и $f(u, s) = f(v, u) = f(t, v) = -1$. Для всех остальных рёбер $(u, v) \in E'$ положим $f(u, v) = 0$. Неформально говоря, каждое ребро $(u, v) \in M$ соответствует единичному потоку по пути $s \to u \to v \to t$. Никакие два таких пути не содержат общих вершин (кроме истока и стока) или рёбер.

Чтобы проверить, что f является потоком, достаточно заметить, что f представим в виде суммы потоков по этим путям. Поток через разрез $(L\cup\{s\}, R\cup\{t\})$ равен $|M|$, поэтому по лемме 27.5 значение потока $|f|$ равно $|M|$.

Докажем обратное. Пусть $|f|$ — целочисленный поток в G'; его значения могут быть равны 0 или 1, так как пропускная способность рёбер ограничена единицей. Определим

$$M = \{(u, v)|u \in L, v \in R, f(u, v) = 1\}.$$

Докажем, что M — паросочетание. В самом деле, из одной вершины u не могут выходить два ребра (u, v') и (u, v''), по которым *поток равен* 1*, так как входящий в и поток не превосходит* 1*. По аналогичным причинам в любую вершину* v *входит не более одного ребра с единичным потоком.*

Чтобы убедиться, что $|M| = |f|$*, заметим, что* $|M|$ *есть поток через разрез* $(L \cup \{s\}, R \cup \{t\}$ *(по каждому ребру идёт потом* 1*, а число рёбер есть* M*).*

Лемма сводит задачу о максимальном паросочетании к задаче о максимальном целочисленном потоке. Требования целочисленности у нас раньше не было, но оказывается, что метод Форда–Фалкерсона всегда даёт целочисленный максимальный поток, если только пропускные способности всех рёбер целые, так что специально заботится о целочисленности не надо.

Теорема 27.11 (Теорема о целочисленном потоке)

Если пропускные способности всех рёбер — целые числа, то максимальный поток, найденный алгоритмом Форда–Фалкерсона, будет целочисленным.

Доказательство

На каждом шаге (при каждом добавлении потока по дополняющему пути) поток остаётся целочисленным. (Подробное доказательство мы оставляем читателю в качестве упр.27.3-2.)

Следствие 27.12

Число рёбер в максимальном паросочетания M *в двудольном графе* G *равно значению максимального потока в сети* G'*.*

Доказательство

По теореме 27.11 можно заменить слова "максимального потока" на "максимального целочисленного потока", после чего сослаться на лемму 27.10.

Таким образом, чтобы найти максимальное паросочетание в двудольном графе G*, нам достаточно применить метод Форда–Фалкерсона и найти максимальный поток в соотвествующей сети* G'*. Оценим время работы такого алгоритма. Никакое паросочетание в двудольном графе не может содержать более* $\min(L, R) = O(V)$ *рёбер, поэтому значение максимального потока в* G' *равно* $O(V)$*. Следовательно, время работы алгоритма Форда–Фалкерсона равно* $O(VE)$ *(см. анализ алгоритма Форда–Фалкерсона в разделе 27.2).*

Упражнения

27.3-1

Примените алгоритм Форда-Фалкерсона к сети рисунка 27.9(b). Как будут выглядеть остаточные сети после каждого шага? (Пронумеруйте вершины сверху вниз отдельно в L *и в* R*; на каждом шаге выбирайте наименьший в смысле лексикографического порядка дополняющий путь.)*

27.3-2

Докажите теорему 27.11.

27.3-3 Пусть $G = (V, E)$ *— двудольный граф и* G' *— сот-*

ветствующая сеть. Оценить сверху длину любого дополняющего пути в G', найденного при работе процедуры FORD–FULKERSON.

*27.3-4**

Полным (совершенным) *паросочетанием (perfect matching) называется паросочетание, в которое входят все вершины. Пусть $G = (V, E)$ — неориентированный двудольный граф, в котором доли L и R имеют поровну элементов. Для каждого $X \subseteq V$ определим* множество соседей *(neighborhood of) X формулой*

$$N(X) = \{y \in V | (x, y) \in E \text{ для некоторого } x \in X,$$

(соседями являются вершины, соединённые ребром с некоторым элементом множества X). Докажите теорему Холла *(Hall's theorem): полное паросочетание существует тогда и только тогда, когда для любого $A \subset L$ выполнено $|A| \leqslant |N(A)|$.*

*27.3-5**

*Назовем двудольный граф $G = (V, E)$ с долями L и R d-*регулярным *(d-regular), если степень каждой вершины $v \in V$ равна d. (Заметим, что в таком случае $|L| = |R|$.) Докажите, что для d-регулярного графа всегда существует полное паросочетание.*

27.4 Алгоритм проталкивания предпотока

В этом разделе мы излагаем метод "проталкивания предпотока". Наиболее быстрые из известных алгоритмов для задачи о максимальном потоке используют именно его. Он применим и к другим задачам, например к задаче о потока наименьшей стоимости. В этом разделе, следуя Гольдбергу, мы опишем " общий" метод проталкивания предпотока. Уже простейшая его реализация требует всего лишь $O(V^2E)$ шагов и опережает алгоритм Эдмондса-Карпа ($O(VE^2)$). В разделе 27.5 мы покажем, как улучшить оценку до $O(V^3)$.

В отличие от алгоритма Эдмондса-Карпа, мы не просматриваем всю всю остаточную сеть на каждом шаге, а действуем локально в окрестности одной вершины. Кроме того, мы не требуем, выполнения закона сохранения потока в процессе работы алгоритма, довольствуясь выполнением свойств предпотока. Мы будем называть предпотоком *(preflow) функцию $f : V \times V \to \mathbb{R}$, которая кососимметрична, удовлетворяет ограничениям, связанным с пропускными способностями, а также такому и ослабленному закону сохранения: $f(V, u) \geqslant 0$ для всех вершин $u \in V \setminus \{s\}$. Таким образом, в каждой вершине u (кроме истока) есть некоторый неотрицательный* избыток *(excess flow)*

$$e(u) = f(V, u). \tag{27.8}$$

Вершину (отличную от истока и стока) с положительным избытком назовём переполненной *(overflowing). (Пользуясь метафорой нефтепровода, можно сказать, что приходящую нефть не успевают откачивать, и избыток нефти сливается.)*

В этом разделе мы объясним идею метода и опишем две основные операции: "проталкивание" предпотока и "подъём" вершины. После этого докажем правильность общего алгоритма проталкивания предпотока и оценим время его работы.

Мотивировка

В методе Форда–Фалкерсона мы в каждый момент имеем дело с потоком жидкости по трубам от истока к стоку; на каждом шаге мы увеличиваем этот поток, находя дополняющий путь. Жидкость никуда не проливается по дороге.

В алгоритмах проталкивания предпотока избыток жидкости в каждой вершине (месте соединения труб) сливается. Кроме того, важную роль играет целочисленный параметр, который будет называться высотой вершины — мы будем воображать, что в процессе работы алгоритма вершина может подниматься вверх. Высота вершины определяет, куда мы стараемся направить избыток жидкости: хотя (положительный) поток жидкости может идти и снизу вверх, увеличивать его в такой ситуации нельзя. (Подробности см. ниже.)

Высота истока всегда равна $|V|$*, а стока — нулю. Все остальные вершины изначально находятся на высоте* 0*, и со временем поднимаются. Для начала мы отправляем из истока вниз столько жидкости, сколько нам позволяют пропускные способности выходящих из истока труб (это количество равно пропускной способности разреза* $(s, V \setminus s)$*). Возникающий (в соседних с истоком вершинах) избыток жидкости сперва просто выливается, но затем он будет направлен дальше.*

Рассматривая какую-либо вершину u *в ходе работы алгоритма, мы можем обнаружить, что в ней есть избыток жидкости, но что все трубы, по которым ещё можно отправить жидкость из* u *куда-то (все ненасыщенные трубы) ведут в вершины той же или большей высоты. В этом случае мы можем выполнить другую операцию, называемую " подъёмом" вершины* u*. После этого вершина* u *становится на единицу выше самого низкого из тех её соседей, в которого ведёт ненасыщенная труба — другими словами, мы поднимаем* u *ровно настолько, чтобы появилась ненасыщенная труба, ведущая вниз.*

В конце концов мы добьёмся того, что в сток приходит максимально возможное количество жидкости (для данных пропускных способностей труб). При этом предпоток может ещё не быть потоком (избыток жидкости сливается). Продолжая подъём вершин (которые могут стать выше истока), мы постепенно отправим избыток обратно в исток (что означает сокращение *потока жидкости от истока) — и превратим предпоток в поток (который окажется максимальным).*

Основные операции

Итак, алгоритм проталкивания предпотока использует две основные операции: проталкивание потока из вершины в соседнюю и подъём вершины. Дадим точные определения.

Пусть $G = (V, E)$ *— сеть с истоком* s *и стоком* t*, а* f *— предпоток в* G*. Функция* $h : V \to \mathbb{N}$ *называется* высотной функцией *(height function) для предпотока* f*, если* $h(s) = |V|$, $h(t) = 0$ *и*

$$h(u) \leqslant h(v) + 1$$

для любого остаточного ребра $(u, v) \in E_f$*. Следующая лемма даёт очевидную переформулировку этого условия:*

Лемма 27.13

Пусть f *— предпоток в сети* $G = (V, E)$ *и* h *— высотная функция. Тогда если для вершин* $u, v \in V$*, выполнено* $h(u) > h(v) + 1$*, то остаточная сеть не содержит ребра* (u, v) *(”по круто идущим вниз трубам идёт максимально возможный поток”.)*

Теперь определим основные операции.

Процедура Push*(*u, v*) применима, если вершина* u *переполнена (то есть* $c_f(u, v) > 0$*) и если* $h(u) = h(v) + 1$*. При этом поток из вершины* u *в её соседа* v *растёт — его увеличение ограничено избытком жидкости в* u *и остаточной пропускной способностью ребра* (u, v)*.*

```
\textsc{Push}($u,v$)
1  $\triangleright$ дано: вершина $u$ переполнена,
            $c_f(u,v)>0$ и $h(u)=h(v)+1$.
2  $\triangleright$ надо: протолкнуть
$d_f(u,v)=\min(e[u],c_f(u,v)$  единиц потока из  $u$ в  $v$.
3  $d_f(u,v)\leftarrow\min(e[u],c_f(u,v)$
4  $f[u,v]\leftarrow f[u,v]+d_f(u,v)$
5  $f[v,u]\leftarrow  -f[u,v]$
6   $e[u]\leftarrow e[u]-d_f(u,v)$
7   $e[v]\leftarrow e[v]+d_f(u,v)$
```

Мы предполагаем, что избыток в вершине равен $e[u] > 0$*, и что остаточная пропускная способность ребра* (u, v) *также положительна. Поэтому мы можем направить* $\min(e[u], c_f(u, v)) > 0$ *единиц потока из* u *в* v *(мы вычисляем эту величину в строке 3), не превысив пропускной способности и не сделав избыток отрицательным. Следовательно, если функция* f *останется предпотоком (если он им была). Мы изменяем* f *в строках 4-5 и* e *в строках 6-7.*

Условие $h(u) = h(v) + 1$ *гарантирует, что мы направляем дополнительный поток лишь по рёбрам, идущим вниз с единичной* *разницей высот. Впрочем, более крутые рёбра уже насыщены и так (лемма 27.13)*

Операция Push *называется* проталкиванием *из вершины u в вершину v (применённым к вершине u). Проталкивание называется* насыщающим *(saturating), если в результате ребро (u,v) становится* насыщенным *(saturated), то есть если $c_f(u,v)$ обращается в нуль (ребро исчезает из остаточной сети); в противном случае проталкивание считают* ненасыщающим *(nonsaturating).*

Процедура Lift*(u) поднимает переполненную вершину u на максимальную высоту, которая допустима по определению высотной функции. Посмотрим на соотношение высот вершины и её соседей в остаточной сети. По определению высотной функции высота вершины превосходит высоту соседа в остаточной сети не более чем на 1. Если есть сосед, который на единицу ниже, то можно выполнить проталкивание (но нельзя выполнить подъём). Если все соседи не ниже, то проталкивание выполнить нельзя, а подъём — можно, после чего возможно проталкивание. Вот как выглядит процедура* подъёма *(lifting):*

```
\textsc{Lift}($u$)
1 $\triangleright$ дано: вершина $u$ переполнена; для любого
    ребра $(u,v)\in E_f$ выполнено неравенство $h(u)\le h(v)$.
2  $\triangleright$ надо:  увеличить $h[u]$, подгото-
вляя проталкивание
        из вершины $u$
3   $h[u]\leftarrow  1+\min\{h[v]|(u,v)\ in E_f\}$
```

Заметим, что если вершина u переполнена, то в E_f найдётся по крайней мере одно ребро, выходящее из u (минимум в строке 3 берётся по непустому множеству). Чтобы доказать это, вспомним, что $f[V,u] = e[u] > 0$, поэтому существует по крайней мере одна такая вершина v, для которой $f(v,u) > 0$. Получаем

$$c_f(u,v) = c(u,v) - f[u,v] = c(u,v) + f[v,u] > 0,$$

а это означает, что $(u,v) \in E_f$. (Если в вершине жидкость выливается, то она откуда-то приходит, и есть резерв, состоящий в уменьшении этого прихода.)

Общая схема алгоритма

Алгоритм начинается с вызова Initialize-Preflow, *задающего начальный предпоток:*

$$f[u,v] = \begin{cases} c(u,v) & \text{если } u = s, \\ -c(v,u) & \text{если } v = s, \\ 0 & \text{в остальных случаях.} \end{cases} \qquad (27.9)$$

```
\textsc{Initialize-Preflow}($G,s$)
for (для) каждой вершины $u\in V[G]$
 do $h[u]\leftarrow  0$
    $e[u]\leftarrow  0$
for (для) каждого ребра $(u,v)\in E[G]$
    do $f[u,v]\leftarrow  0$
     $f[v,u]\leftarrow  0$
      $h[s]\leftarrow |V[G]|$
for (для) каждой вершины $u\in Adj[s]$
    do $f[s,u]\leftarrow c(s,u)$
     $f[u,s]\leftarrow  -c(s,u)$
     $e[u]\leftarrow c(s,u)$
```

В массиве h *хранятся высоты, в массиве* e *— избытки,* $c(u,v)$ *— пропускные способности (считаем, что они заданы так, что вычисление* $c(u,v)$ *требует времени* $O(1)$*). Поток записывается в массив* f*.*

Поток по каждому ребру, выходящему из истока s*, становится равным пропускной способности этого ребра. По остальным рёбрам поток равен* 0*. В каждой смежной с истоком вершине* v *появляется избыток* $e[v]=c(s,v)$*. Начальная высота задается формулой*

$$h[u]=\begin{cases} |V|, \text{ если } u=s, \\ 0, \text{ в противном случае.}\end{cases}$$

Это действительно высотная функция, так как рёбра (u,v)*, для которых* $h[u]>h[v]+1$*, выходят только из истока* ($u=s$)*, но эти рёбра насыщены и их нет в остаточной сети.*

Программа Generic-Preflow-Push *даёт общую схему алгоритма, основанного на проталкивании предпотока.*

```
\textsc{Generic-Preflow-Push}
\textsc{Initialize-Preflow}
 while (пока)  возможны операции подъёма или проталкивания
    do выполнить одну из этих операций
```

Следующая лемма показывает, что пока есть хоть одна переполненная вершина, какая-то из операций подъёма или проталкивания возможна.

Лемма 27.14 (В переполненной вершине возможно либо проталкивание, либо подъём)

Пусть f *— предпоток в сети* $G=(V,E)$*. Пусть* h *— высотная функция для* f *и вершина u переполнена. Тогда в u возможно либо проталкивание, либо подъём.*

Доказательство

Поскольку h — высотная функция, то $h(u) \leqslant h(v)+1$ для любого остаточного ребра (u,v). Если в u невозможно проталкивание, то для всех остаточных рёбер (u,v) выполнено неравенство $h(u) < h(v)+1$, из чего следует, что $h(u) \leqslant h(v)$, и в вершине u возможен подъём.

Корректность метода

Корректность метода проталкивания предпотока мы докажем в два этапа. Сначала мы докажем, что если алгоритм остановится, то предпоток f в этот момент будет максимальным потоком. Затем мы докажем, что алгоритм действительно остановится. Начнём с такого замечания (очевидного следующего из описания процедуры подъёма).

Лемма 27.15 (Высота вершины не убывает)

При исполнении программы Generic-Preflow-Push *высота $h[u]$ любой вершины $u \in V$ может только возрастать (при каждом подъёме этой вершины по меньшей мере на единицу).*

Лемма 27.16

Во время выполнения программы Generic-Preflow-Push *функция h остаётся высотной функцией.*

Доказательство

Посмотрим, что происходит при проталкивании и при подъёме.

При подъёме вершины u мы заботимся о том, чтобы выходящие из u остаточные рёбра не нарушали определение высотной функции. Что же касается входящих рёбер, то с ними не может быть проблем, так как высота вершины u только возрастает.

Рассмотрим теперь процедуру Push*(u,v). Легко понять, что круто идущие внизх ненасыщенные рёбра появиться не могут. Более формально, эта процедура может добавить ребро (v,u) в E_f, а также удалить ребро (u,v) из E_f. В первом случае $h[v]=h[u]-1$ и h остаётся высотной функцией. Во втором случае удалению ребра сопутствует отмена соответствующего ограничения и h снова остаётся высотной функцией.*

Докажем одно важное свойство высотной функции.

Лемма 27.17

Пусть $G=(V,E)$ — сеть с истоком s и стоком t. Пусть f — предпоток в G, а h — высотная функция для f. Тогда в остаточной сети G_f не существует пути из истока в сток.

Доказательство

Пусть это не так и такой путь существует. Устраняя циклы, можно считать, что он простой и потому длина его меньше $|V|$. При этом высота падает от $|V|$ до нуля. Следовательно, в пути есть ребро, где высота падает по крайней мере на 2 — а такое ребро не может входить в остаточную сеть.

Теорема 27.18 (Корректность метода проталкивания предпотока)

Если программа Generic-Preflow-Push, *применённая к сети $G = (V, E)$ с истоком s и стоком t останавливается, то получающийся предпоток f, будет максимальным потоком для G.*

Доказательство

Лемма 27.14 гарантирует, что в момент остановки переполненных вершин в сети нет (избыток в каждой равен нулю). Значит, в этот момент предпоток является потоком. По лемме 27.16 функция h будет высотной функцией, и потому (лемма 27.17) в остаточной сети G_f нет пути из s в t. По теореме о максимальном потоке и минимальном разрезе поток f максимален.

Анализ метода

Чтобы убедить, что алгоритм проталкивания предпотока останавливается, укажем верхние границы отдельно для числа подъёмов, насыщающих и ненасыщающих проталкиваний. После этого станет ясно, что время работы алгоритма есть $O(V^2E)$.

Начнём с такой важной леммы:

Лемма 27.19

Пусть $G = (V, E)$ — сеть с истоком s и стоком t, а f — предпоток в G. Тогда для любой переполненной вершины u найдется простой путь из u в s в остаточной сети G_f.

Доказательство.

Жидкость, сливаемая в вершине u, попадает туда из истока s по какому-то пути: существует путь из s в u, по рёбрам которого идёт положительный поток. (Формально можно рассуждать так: рассмотрим множество U тех вершин, из которых в u можно пройти по рёбрам с положительным потоком. Если среди них нет истока, то в U ни по каким рёбрам жидкость не входит. Отсюда следует, что $e(U)$ (сумма всех избытков вершин в U), равная $f(V, U) = f(V \setminus U, U) + f(U, U) = f(V \setminus U, U) \leqslant 0$, так что все изюытки равны 0.)

Итак, возьмём путь из s в u, по всем рёбрам которого идёт положительный поток, и обратим его рёбра. Обратные рёбра входят в остаточную сеть. Для доказательства утверждения леммы остаётся удалить циклы (если они есть).

Следующая лемма ограничивает высоту вершины, и тем самым число возможных подъёмов.

Лемма 27.20

При исполнении программы Generic-Preflow-Push *высота любой вершины $v \in V$ никогда не превзойдёт $2|V| - 1$.*

Доказательство

По определению высоты $h[s] = |V|$ и $h[t] = 0$. Подъём применим только к переполненным вершинам — посмотрим, на какой высоте они могут быть. Пусть u — переполненная вершина. По лемме 27.19, в G_f существует простой путь p из этой вершины в s. По определению высотной функции, высота не может убывать *более чем на 1 вдоль рёбер сети* G_f*, а высота конечной вершины пути (т.е.* s*) равна* $|V|$*. Путь (будучи простым) содержит не более* $|V|-1$ *рёбер, так что высота его начала не превосходит* $2|V|-1$*.*

Следствие 27.21 (Оценка числа подъёмов)

При исполнении программы Generic-Preflow-Push *общее число операций подъёма не превосходит* $2|V|^2$*.*

Доказательство

Высота вершины при подъёме увеличивается, но не может стать больше $2|V|-1$*, поэтому любую вершину* $v \in V \setminus \{s,t\}$ *можно поднять самое большее* $2|V|-1$ *раз. Всего таких вершин* $|V|-2$*, поэтому общее число подъёмов не превосходит* $(2|V|-1)(|V|-2|) < 2|V|^2$*.*

Лемма 27.22 (Оценка числа насыщающих проталкиваний) При исполнении программы Generic-Preflow-Push *количество насыщающих проталкиваний не превосходит* $2|V||E|$*.*

Доказательство

Рассмотрим насыщающие проталкивания между вершинами $u, v \in V$ *(в обе стороны). Если хотя бы одно проталкивание было, то хотя бы одно из рёбер* (u,v) *или* (v,u) *принадлежит* E*. Пусть имело место насыщающее проталкивание из* u *в* v*. После него ребро* (u,v) *исчезло из остаточной сети* G_f*. Для того, чтобы это ребро появилось, необходимо протолкнуть поток из* v *в* u*, но этого нельзя сделать, пока не будет выполнено* $h[v] = h[u]+1$*, т.е.* $h[v]$ *необходимо увеличить по крайней мере на 2.*

Посмотрим на значение суммы $h[u]+h[v]$ *в моменты насыщающих проталкивания между* u *и* v*. Заметим, что проталкивание возможно, только если высоты вершин* u *и* v *отличаются на единицу. Поэтому первая сумма не меньше* 1*, а последняя сумма не больше* $(2|V|-1)+2(|V|-2) = 4|V|-3$*. Две соседние суммы отличаются по крайней мере на* 2*. Таким образом, всего имеется не более* $((4|V|-3)-1)/2+1 = 2|V|-1$ *насыщающих проталкиваний. (Мы добавили единицу, чтобы учесть и первое, и последнее проталкивание.). Следовательно, общее число насыщающих проталкиваний (для всех рёбер) не превосходит* $(2|V|-1)|E| < 2|V||E|$*.*

Лемма 27.23 (Оценка числа ненасыщающих проталкиваний)

При исполнении программы Generic-Preflow-Push *число ненасыщающих проталкиваний не превосходит* $4|V|^2(|V|+|E|)$*.*

Доказательство

Назовём потенциалом сумму высот переполненных вершин, и будем смотреть, как меняется потенциал (обозначим его Φ*) в ходе исполнения программы. Изначально* $\Phi = 0$*. Подъем произвольной вершины* u *увеличивает* Φ *не более, чем на* $2|V|$ *(высота вершины не превосходит* $2|V|$*, лемма 27.20). Насыщающее проталкивание из некоторой вершины* u *в некоторую вершину* v *может увеличить потенциал лишь за счёт появления новой переполнен-*

ной вершины v (так будет, если v — не сток и не исток), то есть не более чем на $2|V|$. *Наконец, ненасыщающее проталкивание из u в v уменьшает* Φ *по крайней мере на единицу, так как вершина u перестаёт быть переполненной и слагаемое* $h[u]$ *исчезает, а появиться может лишь слагаемое* $h[v]$ *(если вершина v не сток, не исток и не была переполненной), которое на единицу меньше.*

Таким образом, общая сумма, на которую увеличивается Φ *во время работы программы, не превосходит* $(2|V|)(2|V|^2) + (2|V|)(2|V||E|) = 4|V|^2(|V| + |E|)$ *(мы используем следствие 27.21 и лемму 27.22) Но* $\Phi \geqslant 0$, *поэтому общая сумма, на которую* Φ *уменьшится, и тем самым общее количество ненасыщающих проталкиванийю не превосходит* $4|V|^2(|V| + |E|)$.

Теорема 27.24

Общее число операций подъёма и проталкивания при исполнении программы Generic-Preflow-Push *на сети* $G = (V, E)$ *равно* $O(V^2E)$.

Доказательство

Применяем следствие 27.21 и леммы 27.22, 27.23.

Следствие 27.25

Алгоритм, основанный на проталкивании предпотока, можно реализовать так, чтобы на сети $G = (V, E)$ *время его работы было* $O(V^2E)$.

Доказательство

Легко видеть (упр. 27.4-1), что можно выполнить подъём за время $O(V)$ *и проталкивание за время* $O(1)$, *что и даёт требуемую оценку.*

Упражнения

27.4-1

Как реализовать алгоритм проталкивания предпотока так, чтобы на подъём уходило время $O(V)$, *и на проталкивание* $O(1)$? *(При этом общее время будет* $O(V^2E)$.*)*

27.4-2

Докажите, что алгоритм проталкивания предпотока, на все $O(V^2)$ *подъёмов тратит* $O(VE)$ *времени.*

27.4-3

Допустим, мы нашли максимальный поток в сети G методом проталкивания предпотока. Как теперь быстро найти минимальный разрез?

27.4-4

Как найти максимальное паросочетание в двудольном графе, используя метод проталкивания предпотока? Каково время работы вашего алгоритма?

27.4-5

Пусть все пропускные способности рёбер сети $G = (V, E)$ — *целые числа от 1 до k. Оцените в терминах* $|V|$, $|E|$ *и k время ра-*

боты алгоритма проталкивания предпотока. (Указание: сколько ненасыщающих проталкиваний можно применить к ненасыщенному ребру, прежде чем оно станет насыщенным?)

27.4-6

Докажите, что строку 7 процедуры Initialize-Preflow *можно заменить строкой*

$$h[s] \leftarrow |V[G]| - 2,$$

не нарушая корректности и не меняя асимптотики времени работы алгоритма.

27.4-7

Обозначим через $\delta_f(u,v)$ *расстояние (количество рёбер) от вершины* u *до вершины* v *в остаточной сети* G_f*. Покажите, что во время выполнения программы* Generic-Preflow-Push *остаётся верными следующие утверждения: если* $h[u] < |V|$*, то* $h[u] \leqslant \delta_f(u,t)$*; если* $h[u] \geqslant V$*, то* $h[u] \leqslant \delta_f(u,s)$*.*

*27.4-8**

Как и в предыдущем упражнении, $\delta_f(u,v)$ *обозначает расстояние от вершины* u *до вершины* v *в остаточной сети* G_f*. Как изменить алгоритм проталкивания предпотока, чтобы во время его работы оставались верными такие утверждения: если* $h[u] < |V|$*, то* $h[u] = \delta_f(u,t)$*; если* $h[u] \geqslant V$*, то* $h[u] = \delta_f(u,s)$*. (Дополнительные действия должны укладываться в* $O(VE)$ *операций.)*

27.4-9

Покажите, что число ненасыщающих проталкиваний, выполненных программой Generic-Preflow-Push *на сети* $G = (V,E)$*, не превосходит* $4|V|^2|E|$ *(если* $|V| \geqslant 4$*).*

27.5 Алгоритм поднять-и-в-начало

Пользуясь методом проталкивания предпотока, мы применяли операции подъёма и проталкивания в более или менее произвольном порядке. Более продуманный порядок выполнения этих операций позволяет уменьшить время работы алгоритм (по сравнению с оценкой $O(V^2E)$ *из следствия 27.25). В этом разделе мы рассмотрим алгоритм "поднять-и-в-начало" (lift-to-front algorithm), использующий эту идею; время его работы есть* $O(V^3)$*, что асимптотически по крайней мере не хуже, чем* $O(V^2E)$*.*

Алгоритм "поднять-и-в-начало" хранит все вершины сети в виде списка. Алгоритм просматривает этот список, начиная с головы. и находит в нем переполненную вершину u*. Затем алгоритм "обслуживает" эту вершину, применяя к ней операции подъёма и проталкивания до тех пор, пока избыток не станет равным нулю. Если для этого вершину пришлось поднять, её пере-*

мещают в начало списка (отсюда и название алгоритма), и просмотр списка начинается вновь.

При анализе алгоритма пользуемся понятием допустимого ребра — ребра остаточной сети, по которому возможно проталкивание. Сначала мы изучим некоторые их свойства и рассмотрим процесс " обслуживания" вершины.

Допустимые рёбра

Пусть f — предпоток в сети $G = (V, E)$, а h — высотная функция. Назовем ребро (u, v) допустимым *(admissible), если оно входит в остаточную сеть ($c_f(u, v) > 0$) и $h(u) = h(v)+1$ Остальные рёбра мы будем называть* недопустимыми *(inadmissible). Обозначим через $E_{f,h}$ множество допустимых рёбер; сеть $G_{f,h} = (V, E_{f,h})$ назовём* сетью допустимых рёбер *(admissible network). Она состоит из рёбер, по которым возможно проталкивание. Поскольку вдоль допустимого ребра высота уменьшается, имеет место такая лемма:*

Лемма 27.26 (Допустимые рёбра образуют ациклический граф)

Пусть f — предпоток в сети $G = (V, E)$; пусть h — высотная функция. Тогда сеть допустимых рёбер $G_{f,h} = (V, E_{f,h})$ не содержит циклов.

Посмотрим, как изменяют сеть допустимых рёбер операции подъёма и проталкивания.

Лемма 27.27

Пусть f — предпоток в сети $G = (V, E)$ и h — высотная функция. Пусть (u, v) — допустимое ребро и вершина и переполнена. Тогда по (u, v) возможно проталкивание. В результате выполнения этой операции новые допустимые рёбра не появляются, но ребро (u, v) может стать недопустимым.

Доказательство

В результате проталкивания в остаточной сети может появиться только ребро (v, u). Поскольку ребро (u, v) допустимо, то $h(v) = h(u) - 1$, и потому ребро (v, u) недопустимо. Если проталкивание оказывается насыщающим, то в результате $c_f(u, v) = 0$ и ребро (u, v) исчезает из остаточной сети (и становится недопустимым).

Лемма 27.28

Пусть f — предпоток в сети $G = (V, E)$ и h — высотная функция. Если вершина и переполнена и из нее не выходит допустимых рёбер, то возможен подъём вершины и. После подъёма появится по крайней мере одно допустимое ребро, выходящее из вершины и и не будет рёбер, входящих в и.

Доказательство

Как мы видели (лемма 27.14), в переполненной вершине и возможно либо проталкивание, либо подъём. Так как из и допустимые рёбра не выходят, то проталкивание в ней невозможно, и возможен подъём. При этом высота вершины увеличивается *так, что проталкивание становится возможным, то есть появляется допустимое ребро.*

Проверим второе утверждение леммы. Предположим, что после подъёма имеется допустимое ребро (v,u). *Тогда* $h(v)=h(u)+1$ *— а до подъёма было выполнено* $h(v)>h(u)+1$. *По определению высотной функции ребро* (u,v) *должно быть насыщенным (до и после подъёма — подъём не меняет потоков), и потому не входит в остаточную сеть и не является допустимым.*

Списки соседей

Алгоритм "поднять-и-в-начало" использует специальный способ хранения рёбер сети $G=(V,E)$. *Именно, для каждой вершины* $u\in V$ *имеется односторонне связвнный* список соседей *(neighbor list)* $N[u]$. *Вершина* v *фигурирует в этом списке, если* $(u,v)\in E$ *или если* $(v,u)\in E$. *Таким образом, список* $N[u]$ *содержит все вершины* v, *для которых* (u,v) *имеет шанс появиться в остаточной сети. Первый элемент этого списка обозначается* $head[N[u]]$; *следующий за вершиной* v *сосед —* $next{-}neighbor[v]$. *Если вершина* v *— последняя в списке, то* $next-neighbor[v]=$NIL.

Порядок в списке соседей может быть произвольным; он не меняется в ходе работы (всякий раз алгоритм просматривает список соседей в одном и том же порядке). Для каждой вершины u *хранится указатель* $current[u]$ *на очередной элемент списка* $N[u]$. *Изначально* $current[u]$ *установлен на* $head[N[u]]$.

Обработка переполненной вершины

Обработка переполненной вершину u *состоит в том, что её* разряжают *(discharge), проталкивая весь избыток потока в соседние по допустимым рёбрам. Иногда для этого необходимо создать новые допустимые рёбра, подняв вершину* u.

```
\textsc{Discharge}($u$)
1  while $e[u]>0$
2  do $v\leftarrow  current[u]$
3   if $v=$\textsc{nil}
4           then  \textsc{Lift($u$)}
5                $current[u]\leftarrow head[N[u]]$
6              elseif $c_f(u,v)>0$ and $h[u]=h[v]+1$
7              then  \textsc{Push($u,v$)}
8          else $current[u]\leftarrow next-neighbor[v]$
```

На рис. 27.10 показаны несколько итераций цикла **while** *(строки 1–8).*

Каждая итерация цикла **while** *производит одно из трёх действий:*

1. Если мы дошли до конца списка ($v=$NIL*), то мы поднимаем вершину* u *(строка 4) и переходим к началу списка* $N[u]$ *(строка 5). Мы увидим (лемма 27.29), что подъём возможен.*

27.10 Разрядка вершины. Требуется 15 повторений цикла **while** в процедуре Discharge, чтобы протолкнуть весь избыток из вершины y. Показаны только соседи вершины y и рёбра, соединяющие их с y. Внутри каждой вершины указан избыток в ней перед соответствующей итерацией; слева указана высота вершины. Справа показан список $N[y]$; выделен сосед $current[y]$. (**a**) Изначально избыток в y равен 19 и $current[y] = s$. Из y не выходят допустимые рёбра, поэтому первые три итерации сдвигают указатель $current[y]$. На четвертом шаге мы дошли до конца списка. Поднимаем вершину y и переходим к началу списка. (**b**) Высота вершины y стала равной 1. Рёбра (y,s) и (y,x) — недопустимые (шаги 5-6), а ребро (y,z) — допустимое, и мы проталкиваем 8 единиц в z (шаг 7; заметим, что указатель $current[y]$ мы при этом не сдвинули). (**c**) Проталкивание на шаге 7 оказалось насыщающим, и на шаге 8 ребро (y,z) становится недопустимым. На шаге 9 мы дошли до конца списка ($current[y] = \text{NIL}$), поднимаем y, переходим к началу. (**d**) Ребро (y,s) недопустимо (шаг 10), но ребро (y,x) допустимо — по нему мы проталкиваем 5 единиц. (**e**) Больше допустимых рёбер нет (шаги 12-13), поэтому ещё раз поднимаем y и возвращаемся к началу списка (шаг 14). (**f**) Проталкиваем 6 единиц в s (шаг 15). (**g**) Избытка в вершине y больше нет, и процедура завершает работу. Заметим, что в этом примере в начале и в конце работы процедуры Discharge указатель $current[y]$ установлен на начало списка, но в общем случае это не так.

2. Если мы не дошли до конца списка и ребро (u,v) — допустимое (проверка в строке 6), то проталкиваем поток из u в v (строка 7).

3. Если мы не дошли до конца списка, но ребро (u,v) — недопустимое, то сдвигаем указатель current$[u]$ *на одну позицию в списке (строка 8).*

Заметим, что при вызове процедуры Discharge *указатель $current[u]$ находится в позиции, "унаследованной" от предыдущего вызова. Последним действием этой процедуры может быть лишь проталкивание: процедура останавливается, если избыток* e$[u]$ *обращается в нуль, но ни подъём вершины, ни сдвиг указателя не меняют эту величину.*

Надо проверить, что процедура Discharge *выполняет подъём и проталкивание тогда, когда это действительно можно сделать по нашим правилам.*

Лемма 27.29

При вызове операции Push*(u,v) в процедуре* Discharge *(строка 7) по ребру (u,v) возможно проталкивание. При вызове операции* Lift*(u) в процедуре* Discharge *(строка 4) возможен подъём вершины u.*

Доказательство

Возможность проталкивания гарантируется проверками в строках 1 и 6, так что первое утверждение очевидно.

Докажем второе утверждение. Для этого (по лемме 27.28) достаточно доказать, что все выходящие из u рёбра недопустимы. Заметим, что при вызовах процедуры Discharge *указатель $current[u]$ перемещается по списку $N[u]$ от его начала $head[N[u]]$ до конца.*

В конце вершину u поднимают и начинается новый проход. Каждый раз, прежде чем сдвинуть указатель с произвольной позиции v мы убеждаемся (строка 6), что ребро (u, v) недопустимо. Таким образом, в конце прохода все выходящие из u рёбра были просмотрены и оказались недопустимыми. Могли ли они затем стать допустимыми (до конца прохода)? По лемме 27.27, проталкивания вообще не создают допустимых рёбер. Их могут породить только операции подъёма. Но вершина u не поднималась (в течение прохода по списку), а подъёмы других вершин создают лишь выходящие из них допустимые рёбра. Поэтому в конце прохода все выходящие из вершины u рёбра недопустимы, поэтому её можно поднять.

Алгоритм "поднять-и-в-начало"

Алгоритм "поднять-и-в-начало" хранит множество $V \setminus \{s, t\}$ вершин (отличных от истока и стока) в виде списка. При этом существенно то, что список этот оказывается "корректно упорядоченным" в следующем смысле: конец любого допустимого ребра находится дальше в списке, чем начало этого ребра (напомним, что допустимые рёбра образуют ациклический граф, лемма 27.26). (Задачу о поиске корректрого упорядочения для произвольного ациклического графа мы называли задачей топологической сортировки, см. раздел 23.4.)

*Следующую в этом списке за u вершину обозначим $next[u]$; если вершина u — последняя в списке, то $next[u] =$*NIL.

```
\textsc{Lift-To-Front}($G,s,t$)
1  \textsc{Initialize-Preflow}($G,s$)
2  $L\leftarrow  V[G]-\{s,t\}$ (в любом порядке)
3  for (для) каждой вершины $u\in V[G]\setminus \{s,t\}$
4    do $current[u]\leftarrow  head[N[u]]$
5  $u\leftarrow head[L]$
6  while $u\ne$\textsc{nil}
7    do $old-height\leftarrow h[u]$
8       \textsc{Discharge}($u$)
9       if $h[u]>old-height$
10         then  переместить $u$ в начало списка $L$
11      $u\leftarrow next[u]$
```

Алгоритм формирует начальный предпоток (строка 1), список L (строка 2) (точно так же, как это делалось раньше). Затем *(строки 3–4) он устанавливает указатели* $current[u]$ *в начало списка соседей каждой вершины* u *(считаем, что для всех вершин* u *списки соседей* $N[u]$ *уже созданы.)*

Работа цикла (строки 6–11, см. также рис. 27.11) происходит так: мы просматриваем все элементы списка L*, начиная с начала (строка 5). Всякий раз мы разряжаем текущую вершину* u *(строка 8). Если при этом высота вершины* u *увеличилась (что определяется сравнением с сохранённым в строке 7 прежним значением), то мы перемещаем её в начало списка (строка 10). После этого мы переходим к следующему элементу списка* L*. Заметим, что если мы переместили* u *в начало списка, то очередным будет элемент, следующий за* u *в её новой позиции.*

Докажем, что алгоритм Lift-To-Front *находит максимальный поток. Для этого убедимся, что его можно рассматривать как реализацию общей схемы проталкивания предпотока. Сначала заметим, что программа применяет подъёмы и проталкивания только там, где они возможны (согласно лемме 29.29). Осталось доказать, что после завершения алгоритма никакие подъёмы или проталкивания невозможны. останавливается, возможных подъёмов и проталкиваний нет.*

Когда мы в последний раз в программе просмотрели список L*, мы разрядили каждую вершину* u*, не подняв её. Как мы вскоре увидим (лемма 27.30), список* L *во время выполнения программы остаётся корректно упорядоченным, то есть конец любого допустимого ребра идёт после его начала. Поэтому процедура* Discharge(u)*, проталкивая поток по допустимым рёбрам, создаёт избыток в вершинах, идущих в списке после* u*, и не трогает вершины, предшествующие* u*, в которых избыток остаётся равным нулю. Таким образом, по завершинию работы избыток в каждой вершине равен нулю (и ни проталкивание, ни подъём невозможны).*

Лемма 27.30

При исполнении алгоритма Lift-To-Front *список* L *остаётся корректно упорядоченным относительно (текущего) графа допустимых рёбер* $G_{f,h} = (V, E_{f,h}$ *после любого числа итераций цикла в строках 6–11.*

Доказательство

Перед первым выполнением цикла допустимых рёбер нет, так как высота истока $h[s]$ *равна* $|V| \geqslant 2$ *(множество* V *содержит по крайней мере исток* s *и сток* t*), а высота остальных вершина равна* 0 *(и единичного перепада высот быть не может).*

Докажем, что свойство топологической упорядоченности сохраняется после каждого выполнения тела цикла. Сеть допустимых рёбер меняют только подъёмы и проталкивания. По лемме 27.27, проталкивания не создают допустимых рёбер. После подъёма произвольной вершины и все рёбра, входящие в эту вершину

27.11 Работа алгоритма Lift-To-Front. (**a**) Начальный поток перед первым выполнением цикла. Из истока s выходит 26 единиц. Справа изображён список L (серым показано текущее значение u). Под каждой вершиной выписан список её соседей (серым выделен сосед $current[u]$). Сначала мы разряжаем вершину x. Она поднимается на высоту 1. Из 12 единиц избытка отправляем 5 в вершину y, а затем оставшиеся 7 в сток t. Высота вершины x увеличилась, так что x помещаем в начало списка L (впрочем, она там и была). (**b**) На следующем шаге разряжаем вершину y, следующую за x. (На рис. 27.10 этот процесс показан подробно.) Высота вершины y увеличилась, и мы перемещаем её в начало списка L. (**c**) Теперь в списке за y следует x и мы разряжаем вершину x, протолкнув 5 единиц потока в сток t. Подъёма не происходит, так что список L не меняется. (**d**) Следующую за x в списке вершину z мы разряжаем, поднимая её до высоты 1 и проталкивая 8 единиц в сток t. Поскольку вершина поднята, она перемещается в начало списка L. (**e**) Переполненных вершин больше нет, поэтому процедура Discharge, последовательно применённая к вершинам y и x, ничего не меняет. Программа Lift-to-front достигает конца списка L и останавливается. Максимальный поток найден.

— недопустимые (лемма 27.28). Поэтому после перемещения u в начало списка порядок элементов в списке становится корректным.

Анализ алгоритма

Покажем, что время работы алгоритма "поднять-и-в-начало" на сети $G = (V, E)$ равно $O(V^3)$. Сначала напомним некоторые уже известные нам факты. Этот алгоритм является реализацией метода проталкивания предпотока, поэтому действует верхняя оценка $O(V)$ на число подъёмов каждой вершины (следствие 27.21), а общее число подъёмов есть $O(V^2)$. В соответствии с упр. 27.4-2, на все подъёмы уходит время $O(VE)$. Общее число насыщающих проталкиваний есть $O(VE)$ (лемма 27.22).

Теорема 27.31

Время работы программы Lift-To-Front *на сети $G = (V, E)$ равно $O(V^3)$.*

Доказательство

Разобьём время работы программы Lift-To-Front *на периоды между двумя подъёмами. Тогда всего периодов будет (как и подъёмов) $O(V^2)$. Покажем, что за каждый период мы вызываем процедуру* Discharge *$O(V)$ раз. Действительно, за один период число вызовов процедуры* Discharge *(поскольку внутри периода мы не поднимаем вершину, список остаётся неизменным) не превосходит длины списка, и тем самым $|V|$.*

Следовательно, процедура Discharge *вызывается $O(V^3)$ раз* *(строка 8), и время работы программы* Lift-To-Front *равно* $O(V^3)$ *плюс суммарное время, уходящее на выполнение вызовов* Discharge. *Оценим второе слагаемое.*

При каждом повторении цикла в процедуре Discharge *совершается ровно одно из трёх действий: подъём вершины, перемещение указателя и проталкивание предпотока. Оценим количество операций каждого из этих типов.*

Начнём с подъёмов (строки 4–5). На все $O(V^2)$ *подъёмов требуется время* $O(VE)$ *(упр. 27.4-2).*

Оценим количество перемещений указателя $current[u]$ *(строка 8). Обозначим через* $degree(u)$ *степень вершины* u. *На каждый подъём вершины* u *приходятся* $O(degree(u))$ *перемещений указателя, поэтому всего производится* $O(V \cdot degree(u))$ *перемещений для каждой вершины. Таким образом, общее число перемещений указателей равно* $O(VE)$ *(по лемме о рукопожатиях, упр. 5.4-1).*

Оценим число проталкиваний (строка 7). Мы уже знаем, что количество насыщающих проталкиваний равно $O(VE)$. *Заметим, что сразу после выполнения ненасыщающего проталкивания процедура* Discharge *прекращает работу, так как избыток потока обращается в ноль. Таким образом, при каждом вызове процедуры выполняется не более одного ненасыщающего проталкивания; всего вызовов* $O(V^3)$, *поэтому ненасыщающих проталкиваний не более* $O(V^3)$.

Таким образом, время работы программы Lift-To-Front *есть* $O(V^3 + VE) = O(V^3)$.

Упражнения

27.5-1

Следуя образцу рис. 27.11, покажите работу программы Lift-To-Front *на сети рис.27.1 (a). Считать, что изначально список* L *имеет вид* $\langle v_1, v_2, v_3, v_4\rangle$, *а списки соседей таковы:*

$$\begin{aligned} N[v_1] &= \langle s, v_2, v_3\rangle \\ N[v_2] &= \langle s, v_1, v_3, v_4\rangle \\ N[v_3] &= \langle v_1, v_2, v_4, t\rangle \\ N[v_4] &= \langle v_2, v_3, t\rangle. \end{aligned}$$

*27.5-2**

Рассмотрим реализацию алгоритма проталкивания предпотока, при которой все переполненные вершины хранятся в виде очереди. Алгоритм разряжает первую вершину из очереди и удаляет её, а если в результате этого появились новые переполненные вершины, они добавляются в конец очереди. Алгоритм останавливается, когда очередь пуста. Покажите, что время работы этого алгоритма равно $O(V^3)$.

27.5-3

Заменим в процедуре Lift *строку 3 строкой:* $h[u] \leftarrow h[u]+1$. *Покажите, что общий алгоритм проталкивания предпотока оста-*

Рисунок 27.11 27.12 Задача о выходе. Начальные точки чёрные, остальные — белые. (**a**) Выход есть (выделены пути, его обеспечивающие). (**b**) Выхода нет.

нется корректным. Как это изменение скажется на работе программы Lift-To-Front*?*

*27.5-4**

Покажите, что алгоритм, построенный по методу проталкивания предпотока, разрешающий вершину самой большой высоты работает время $O(V^3)$.

Задачи

27-1 Задача о выходе

Имеется, граф вершины которого образуют решётку из n строк и n столбцов (рис. 27.12). Обозначим через (i, j) вершину на пересечении i-го столбца и j-ой строки. Все вершины такой решётки ($n \times n$ grid), не считая граничных ($i = 1$, $i = n$, $j = 1$ или $j = n$) имеют четырёх соседей. В задаче о выходе *(escape problem) требуется выяснить, существуют ли m попарно непересекающихся (не имеющих общих вершин) путей от данных $m \leqslant n^2$ начальных точек решётки $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$ к m различным граничным точкам. Например, для задачи о выходе рис. 27.12(a) ответ положителен, а для рис. 27.12 (b) — отрицателен.*

(a) Рассмотрим сеть, в которой пропускные способности имеют не только рёбра, но и вершины. Это означает, что поток, входящий в данную вершину, не может превосходить некоторого числа (пропускной способности вершины). Покажите, что задача о максимальном потоке в такой сети сводится к обычной задаче о максимальном потоке для сети несколько большего размера.

(b) Укажите алгоритм, решающий задачу о выходе. Чему равно время его работы?

27-2 Минимальное покрытие путями

Покрытием путями *(path cover) ориентированного графа $G = (V, E)$ назовём множество путей P с таким свойством: каждая вершина из V принадлежит ровно одному пути из P. Пути могут начинаться и заканчиваться где угодно, и иметь любую длину, в том числе нулевую. Покрытие наименьшим возможным числом путей назовём* минимальным покрытием путями *(minitimum path cover).*

(a) Постройте эффективный алгоритм, находящий минимальное покрытие путями графа $G = (V, E)$. (Указание. Пусть $V = \{1, 2, \ldots, n\}$. Построим граф $G' = (V', E')$, где

$$V' = \{x_0, x_1, \ldots, x_n\} \cup \{y_0, y_1, \ldots, y_n\},$$
$$E' = \{(x_0, x_i) | i \in V\} \cup \{(y_i, y_0) | i \in V\} \cup \{(x_i, y_j\} | (i, j) \in E\},$$

и решим для этого графа задачу о максимальном потоке.)

(b) Правильно ли работает этот алгоритм, если у графа есть циклы? Ответ объясните.

27-3 Эксперименты в космосе

Институт космических исследований планирует серию $E = \{E_1, E_2, \ldots, E_m\}$ *экспериментов в космосе. За результат эксперимента* E_i *спонсоры выплачивают* p_i *долларов. Для этих экспериментов требуются приборы из множества* $I = \{I_1, I_2, \ldots, I_n\}$*; для проведения эксперимента* E_j *необходимо множество* $R_j \subseteq I$ *приборов. Стоимость доставки прибора* I_k *составляет* c_k *долларов. Требуется выяснить, какие эксперименты следует проводить, чтобы прибыль (доход от экспериментов минус стоимость доставки нужных приборов) была максимальной.*

Для решения этой задачи рассмотрим сеть G *с истоком* s*, стоком* t*, вершинами* $I_1, I_2, \ldots, I_n$ *и* $E_1, E_2, \ldots, E_m$*. Сеть имеет также рёбра* (s, I_k) *пропускной способности* c_k *(для всех* $k = 1, 2, \ldots, n$*);* (E_j, t) *пропускной способности* p_j*; (для всех* $j = 1, 2, \ldots, m$*);* (I_k, E_j) *бесконечной пропускной способности, если* $I_k \in E_j$ *(для* $k = 1, 2, \ldots, n$ *и* $j = 1, 2, \ldots, m$*).*

(a) Рассмотрим разрез (S, T) *с конечной пропускной способностью для построенной сети. Пусть* $E_j \in T$*. Покажите, что* $I_k \in T$ *для всех* $I_k \in R_j$*.*

(b) Как, зная все p_j *и пропускную способность минимального разреза сети* G*, найти максимальную прибыль от экспериментов?*

(c) Постройте алгоритм, определяющий, какие эксперименты следует проводить [для получения наибольшей прибыли] и какие для этого нужны приборы. Оцените время его работы как функцию от m*,* n *и* $r = \sum_{j=1}^{m} |R_j|$*.*

27-4 Максимальный поток в изменённой сети

Предположим, нам известен максимальный поток в сети $G = (V, E)$*, в которой все пропускные способности рёбер — целые числа.*

(a) Пусть пропускная способность некоторого ребра $(u, v) \in E$ *увеличилась на единицу. Приведите алгоритм, находящий максимальный поток (для изменённой сети) за время* $O(V + E)$*.*

(b) Пусть пропускная способность некоторого ребра $(u, v) \in E$ *уменьшилась на единицу. Приведите алгоритм, находящий максимальный поток (для изменённой сети) за время* $O(V + E)$*.*

27-5 Масштабирование

Рассмотрим сеть $G = (V, E)$ *с истоком* s *и стоком* t*. Пусть пропускная способность* $c(u, v)$ *любого ребра* $(u, v) \in E$ *— целое число. Положим* $C = \max_{(u,v) \in E} c(u, v)$*.*

(a) Докажите, что пропускная способность минимального раз-

реза сети G не превосходит $C|E|$.

(b) Докажите, что при любом заданном K можно за время $O(E)$ найти дополняющий путь с пропускной способностью не меньше K или установить, что такого пути нет.

Алгоритм Max-Flow-By-Scaling *вычисляет максимальный поток в сети G с помощью масштабирования. Это одна из реализаций метода Форда–Фалкерсона.*

```
\textsc{Max-Flow-By-Scaling}($G,s,t$)
1 $C\leftarrow\max\limits_{(u,v)\in E}{c(u,v)}$
2 сделать поток $f$ нулевым
3 $K\leftarrow 2^{\lfloor\log C\rfloor}$
4  while $K\ge 1$
5    do while существует дополняющий путь $p$ пропускной способности не меньше $K$
6          do дополнить $f$ вдоль $p$
7       $K\leftarrow K/2$
8  return $f$
```

(c) Докажите, что программа Max-Flow-By-Scaling *вычисляет максимальный поток.*

(d) Докажите, что в момент проверки условия цикла в строке 4 пропускная способность минимального разреза остаточной сети не превосходит $2K|E|$.

(e) Докажите, что цикл в строках 5-6 выполняется $O(E)$ раз для любого значения K.

(f) Докажите, что время работы алгоритма Max-Flow-By-Scaling *равно $O(E^2 \log C)$.*

27-6

Двусторонние границы

Рассмотрим сеть $G = (V, E)$. Предположим, что ограничения на поток в этой сети двусторонние. Именно, для потока f и для каждого ребра (u, v) мы требуем, чтобы $b(u, v) \leqslant f(u, v) \leqslant c(u, v)$, где c и b — заданные функции на рёбрах. (Возможно, что эти ограничения противоречивы — тогда потока не существует.)

(a) Пусть f — поток в сети G. Докажите, что $|f| \leqslant c(S, T) - b(T, S)$ для любого разреза (S, T).

(b) Пусть в сети G максимальный поток существует. Докажите, что его значение равно минимальной (по всем разрезам (S, T)) разности $c(S, T) - b(T, S)$.

Рассмотрим сеть $G = (V, E)$ с двусторонними границами с истоком s и стоком t. Обозначим верхнюю и нижнюю границы c и b соответственно. Построим обычную сеть $G' = (V', E')$ с верхней границей (пропускной способностью) c', истоком s' и стоком t', положив

$$V' = V \cup \{s', t'\}$$
$$E' = E \cup \{(s', v)|v \in V\} \cup \{(u, t')|u \in V\} \cup \{(s, t), (t, s)\}.$$

Для рёбер $(u,v) \in E$ положим $c'(u,v) = c(u,v) - b(u,v)$; для всех вершин $u \in V$ положим $c'(s',u) = b(V,u)$, а также $c'(u,t') = b(u,V)$. Кроме того, положим $c'(s,t) = c'(t,s) = \infty$.

(c) Докажите, что поток в сети G существует тогда и только тогда, когда для максимального потока в сети G' все рёбра, входящие в t', насыщены.

(d) Постройте алгоритм, который выясняет, существует ли поток в данной сети с двусторонними границами, и если да, то находит этот поток. Оцените время работы этого алгоритма.

Замечания

Дальнейшее обсуждение сетей и связанных с ними алгоритмов см. в Ивен [65], Лоулер [132], Пападимитриу и Стайглиц [154], Тарьян [188]. Хороший обзор сетевых алгоритмов написали Гольдберг, Тардос и Тарьян [83].

Метод Форда–Фалкерсона изобрели Форд и Фалкерсон [71]. Они же рассмотрели многие задачи, связанные с сетями, в том числе задачу о максимальном потоке и о максимальном паросочетании. Во многих ранних реализациях метода Форда–Фалкерсона дополняющий путь находили с помощью поиска в ширину; Эдмондс и Карп [63] доказали, что в этом случае алгоритм полиномиален. Идею предпотока предложил Карзанов [119]. Метод проталкивания предпотока предложил Гольдберг [82]. Наиболее быстрый (время работы $O(VE\log(V^2E))$) из известных алгоритмов проталкивания предпотока принадлежит Гольдбергу и Тарьяну [85]. Наилучший известный (время работы $O(\sqrt{V}E)$) алгоритм для задачи о максимальном паросочетании предложили Хопкрофт и Карп [101].

28 Сортирующие сети

Во второй части книги изучались сортирующие алгоритмы, реализованные на машинах произвольного доступа (RAM). В этой главе рассматриваются сортирующие алгоритмы, в основу которых положена совершенно иная вычислительная модель — сеть компараторов.

Есть два существенных отличия между RAM-машинами и сетями компараторов. Во-первых, в процессе работы компараторы могут выполнять только операции сравнения. Следовательно, реализовать на них алгоритмы, подобные сортировке подсчётом (см. раздел 9.2), невозможно. Во-вторых, RAM-машины работают последовательно, выполняя одну операцию за один такт работы. Сеть компараторов может работать параллельно, т.е. выполнять одновременно несколько операций. Благодаря этому удаётся отсортировать n чисел за время, существенно меньшее n.

В разделе 28.1 вводятся понятия сети компараторов, сортирующей сети, времени работы сети. В разделе 28.2 мы докажем правило " нуля и единицы", которое упрощает проверку правильности работы сортирующей сети.

Быстрая сортирующая сеть, которую мы построим, представляет собой параллельную реализацию сортировки слиянием (раздел 1.3.1). Сначала (раздел 28.3) строится битонический сортировщик затем (28.4) он слегка модифицируется и получается сливающая сеть, которая соединяет два упорядоченных набора в один. Наконец (28.5) мы соединяем сливающие сети в сортирующую сеть, которая позволяет отсортировать n объектов за время $O(\lg^2 n)$.

28.1 Сети компараторов

Сортирующие сети строятся из компараторов, соединённых проводами. Компаратор (comparator) (рис. 28.1(a)) имеет два входа x, y и два выхода x', y'. Компаратор получает на вход два

Рисунок 28.1 28.1 (a) Компаратор со входами x, y и выходами x', y'. (b) Тот же компаратор в виде вертикальной линии. Показаны входы $x = 7$, $y = 3$ и выходы $x' = 3$, $y' = 7$.

Рисунок 28.2 28.2 (a) Сеть компараторов с 4 входами и 4 выходами, которая является сортирующей. Показаны входные значения в момент 0. (b) Значения на выходах компараторов A и B в момент времени 1. (c) В момент 2 появляются выходные значения компараторов C и D; выходные значения b_1 и b_4 (но не b_2, b_3) определены. (d) Наконец, оставшие два выходных значения появляются на выходах компаратора E.

числа и переставляет их, если они идут в неправильном порядке. Другими словами,

$$\begin{aligned} x' &= min(x, y), \\ y' &= max(x, y). \end{aligned}$$

Договоримся схематично изображать компаратор в виде вертикального отрезка, как на рисунке 28.1(b). Входы рсположены слева, а выходы — справа, при этом верхний выход соответствует минимальному числу, а нижний — максимальному.

Мы считаем, что время работы компаратора (между получением входных данных и выдачей выходных) постоянно и одинаково для всех компараторов.

С помощью проводов (wires) выходы одних компараторов соединяют со входами других. Кроме того, сеть имеет входные и выходные провода. Рассмотрим сеть компараторов с n входными проводами (input wires) $a_1, a_2, \dots, a_n$ и n выходными проводами (output wires) $b_1, b_2, \dots, b_n$. На вход поступает последовательность n чисел (которые мы будем обозначать $\langle a_1, a_2, \dots, a_n \rangle$, как и сами провода); результатом работы будет последовательность $\langle b_1, b_2, \dots, b_n \rangle$.

На рисунке 28.2 приводится пример сети компараторов (comparison network). Сеть с n входами и n выходами изображается n горизонтальными прямыми, которые в некоторых местах соединены вертикальными отрезками — компараторами. Прямая — это не один провод, а несколько — проводом является участок между компараторами. Например, верхняя прямая на рисунке 28.2 состоит из трёх проводов: входного провода a_1 соединенного со входом компаратора A; провода, соединяющего верхний выход компаратора A со входом компаратора C, а также выходного провода b_1, соединенного с верхним выходом компаратора C.

Каждый вход любого из компараторов соединён либо с одним из входных проводов $a_1, a_2, \dots, a_n$, либо с выходом другого компаратора. Каждый выход любого из компараторов соединён либо с входом другого (ровно одного), либо с одним из выходных проводов $b_1, b_2, \dots, b_n$. При этом не может быть циклов: идя по проводам *и проходя компараторы от входов к выходам, нельзя вернуться к исходному компаратору. Это требование позволяет рисовать сети компараторов в виде прямых, как на рисунке 28.2, располагая входы слева, а выходы справа: данные движутся по проводам слева направо.*

Мы считаем, что каждый компаратор выдаёт выходные значения через единицу времени после того, как получит входные значения. Рассмотрим сеть рис. 28.2(a) и представим всебе, что в в момент времени 0 на вход поступают числа $\langle 9, 5, 2, 6\rangle$*. В этот момент компараторы* A *и* B *(и только они) получают входные данные и начинают работать (параллельно). Поэтому в момент времени 1 на их выходах появятся числа (рис. 28.2(b)), которые позволят начать параллельную работу компараторам* C *и* D *(но не* E*). Компаратору* E *придётся дожидаться момента времени 2, когда* C *и* D *закончат работу (рис. 28.2(c)). Ещё через единицу времни выходная последовательность* $\langle 2, 5, 6, 9\rangle$ *будет полностью готова (хотя два из четырёх значений были готовы раньше).*

Как видно из этого примера, время работы сети (прошедшее с момента получения входной последовательности до выдачи выходной последовательности) определяется максимальным числом компараторов, стоящих на пути от входа к выходу. Более формально, мы определяем глубину (depth) провода в сети следующим образом: входные провода имеют нулевую глубину, глубина выходов компаратора (подключённых к нему проводов) равна $max(d_x, d_y) + 1$*, где* d_x *и* d_y *— глубины его входов (подключённых к ним проводов). Это определение корректно, поскольку нет циклов. Назовем глубиной компаратора глубину выходящих из него проводов. Глубиной сети считаем максимальную глубину её выходных проводов (или, что то же самое, максимальную глубину её компараторов). Например, глубина сети на рисунке 28.2 равна* 3 *(компаратор* E *имеет глубину 3). Глубина компаратора равна времени, которое пройдет от начала работы по появления ответов на его выходах, так что время работы сети в целом равно её глубине.*

Размером *(size) сети называют число компараторов в ней.*

Любая сеть компараторов как-то переставляет числа, поданные на кё входы. Её называют сортирующей сетью (sorting network), если для любой входной последовательности получающаяся из неё выходная последовательность монотонно возрастает $(b_1 \leqslant b_2 \leqslant \cdots \leqslant b_n)$*.*

Конечно, далеко не каждая сеть — сортирующая, но сеть рисунка 28.2 таковой является. В самом деле, в момент времени 1 минимальный из входов находится на верхнем выходе одного из компараторов A *и* B*, и в момент времени 2 он кажется на на верхнем выходе компаратора* C*. Аналогичным образом максимальный из входов окажется в этот момент на нижнем выходе*

Рисунок 28.3 Сортирующая сеть, соответствующая сортировке вставками (упражнение 28.1-6).

компаратора D. Оставшиеся два числа упорядочиваются компаратором E в момент времени 3.

Сети компараторов аналогичны алгоритмам сортировки, однако для каждого n мы должны строить свою сеть, в то время как один и тот жэе алгоритм сортировки может работать для последовательностей произвольной длины. В этой главе мы построим семейство Sorter *эффективных сортирущих сетей — для каждого n в нём будет своя сеть, которую мы обозначаем* Sorter$[n]$.

Упражнения.

28.1-1

Каковы будут значения на проводах сети рис. 28.2, если на её вход подаются числа $\langle 9,6,5,2\rangle$?

28.1-2

Пусть n — степень числа 2. Постройте сеть компараторов с n входами и n выходами глубины $\lg n$, которая на находит наименьший и наибольший из входов (и выдаёт их на верхний и нижный провода соответственно).

28.1-3

Карлсон говорит Малышу, что в произвольном месте сортирующей сети можно добавить компаратор и сеть останется сортирующей. Покажите Малышу, что это не так, добавив компаратор в сеть рис. 28.2.

28.1-4

Докажите, что любая сортирующая сеть с n входами имеет глубину не меньше, чем $\lg n$.

28.1-5

Докажите, что любая сортирующая сеть содержит не менее $\Omega(n\lg n)$ компараторов.

28.1-6

Покажите, что сеть на рисунке 28.3 является сортирующей, установив её родство с алгоритмом сортировки вставками (см. раздел 1.1).

28.1-7

Сеть компараторов с n входами и с компараторами представима в виде упорядоченного списка, содержащего c пар натуральных чисел от 1 до n. Паре (i,j) соответствует компаратор, соединяющий i-ую и j-ую прямые. Список перечисляет компараторы слева направо. Используя это представление, придумайте (последовательный) алгоритм, который находит глубину сети за время $O(n+c)$.

28.1-8 $\star$

В сети некоторые элементы являются перевёрнутыми компараторами, выдающими на верхний выход максимальный из входов, а на нижний — минимальный. Как преобразовать её в сеть только из обычных компараторов, эквивалентную исходной?

28.2 Правило нуля и единицы

Правило нуля и единицы *(zero-one principle) говорит, что если сеть компараторов упорядочивает любую последовательность нулей и единиц, то является сортирующей (и упорядочивает любую последовательность чисел (целых, вещественных, или вообще элементов произвольного линейно упорядоченного множества). Тем самым, желая доказать, что построенная нами сеть является сортирующей, можно ограничиться рассмотрением входов из нулей и единиц — иногда это упрощает дело.*

Доказательство использует понятие монотонно возрастающей функции (см. раздел 2.2).

Лемма 28.1 Если сеть компараторов преобразует последовательность $a = \langle a_1, a_2, \ldots, a_n \rangle$ *в последовательность* $b = \langle b_1, b_2, \ldots, b_n \rangle$, *а* f *— монотонно возрастающая функция, то подав на вход сети последовательность* $f(a) = \langle f(a_1), f(a_2), \ldots, f(a_n) \rangle$, *мы получим на выходе* $f(b) = \langle f(b_1), f(b_2), \ldots, f(b_n) \rangle$.

Доказательство. Проверим это сначала для одного компаратора, то есть покажем, что компаратор, получив на вход $f(x)$ *и* $f(y)$, *выдаст на выходы* $f(\min(x,y))$ *и* $f(\max(x,y))$ *(рис. 28.4).*

По определнию, на верхнем выходе будет $\min(f(x), f(y))$, *а на нижнем -* $\max(f(x), f(y))$. *Остаётся заметить, что для возрастающей функции* f *из* $x \leqslant y$ *следует* $f(x) \leqslant f(y)$, *и поэтому*

$$\begin{aligned} min(f(x), f(y)) &= f(min(x,y)), \\ max(f(x), f(y)) &= f(max(x,y)). \end{aligned}$$

Эта лемма показывает, что если применить ко входным значения компаратора x *и* y *монотонную функцию* f, *с его выходными значениями произойдёт то же самое.*

Это свойство верно не только для одного компаратора, но и для любой сети компараторов. Пусть на входы сети поданы значения $a_1, a_2, \ldots, a_n$. *Через некоторое время на всех проводах (в том числе выходных) установятся некоторые значения. Теперь заменим входы на* $f(a_1), f(a_2), \ldots, f(a_n)$. *Что случится со значениями на проводах схемы? Как мы только что видели, выходы*

Рисунок 28.4 Компаратор и монотонная функция f (лемма 28.1)

Рисунок 28.5 (a) Сортирующая сеть рисунка 28.2 для входов $\langle 9,5,3,6\rangle$. (b) Та же сеть после применения монотонной функции $f(x)=\lceil x/2\rceil$ ко всем входам. Значение на каждом проводе получается применением f к старому значению на том же проводе.

компараторов глубины 1 *также заменятся на результат применения к ним функции* f. *Следовательно, с выходами компараторов глубины* 2 *произойдёт то же самое, и так далее. Рассуждая по индукции, мы выдим, что на выходах схемы появятся значения* $f(b_1), f(b_2), \ldots, f(b_n)$, *где* $b_1, b_2, \ldots, b_n$ *— прежние выходные значения. Лемма доказана.*

Рисунок 28.5 иллюстрирует утверждение леммы 28.1 для сортирующей сети рис. 28.2 и функции $f(x)=\lceil x/2\rceil$. *Часть (a) показывает значения на проводах до применения* f *(дублируя рис. 28.2 (d), а часть (b) — после.*

С помощью леммы 28.1 легко доказать следующий замечательный результат.

Теорема 28.2 (Правило нуля и единицы)

Если сеть компараторов с n *входами правильно упорядочивает все* 2^n *возможных последовательностей нулей и единиц, то она является сортирующей, то есть правильно упорядочивает любую числовую последовательность.*

Доказательство.

Пусть это не так, и есть числовая последовательность $\langle a_1, a_2, \ldots, a_n\rangle$, *на которой сеть ошибается. Это означает, что есть элементы* a_i, a_j, *для которых* $a_i < a_j$ *и* a_j *попадает в выходную последовательность раньше* a_i. *Нам надо показать, что существует последовательность нулей и единиц, на которой сеть работает неправильно. Для этого рассмотрим монотонную функцию функцию* f:

$$f(x)=\begin{cases} 0, & x \leqslant a_i, \\ 1, & x > a_i \end{cases}$$

Применим её к входам сети, то есть подадим на вход последовательность нулей и единиц $\langle f(a_1), f(a_2), \ldots, f(a_n)\rangle$. *Согласно лемме 28.1 к выходным значениям также применится функция* f. *При этом* $f(a_j)$ *будет стоять на месте* a_j, *то есть раньше* $f(a_i)$. *Но* $f(a_j)=1$, *а* $f(a_i)=0$. *Теорема доказана.*

Упражнения

28.2-1

Ко всем членам упорядоченной последовательности применили монотонно возрастающую функцию f. *Останется ли последовательность упорядоченной?*

28.2-2

Докажите, что сеть компараторов с n *входами правильно упорядочивает последовательность* $\langle n, n-1, \ldots, 1\rangle$ *тогда и только*

Рисунок 28.6 Сортирующая сеть для 4 чисел

тогда, когда она правильно упорядочивает все $n-1$ *последовательностей нулей и единиц:* $\langle 1,0,\ldots,0\rangle$, $\langle 1,0,\ldots,0\rangle$, $\ldots$, $\langle 1,1,\ldots,1,0\rangle$.

28.2-3

Используя правило нуля и единицы, докажите, что сеть компараторов на рисунке 28.6 — сортирующая.

28.2-4

Сформулируйте и докажите аналог правила нуля и единицы для алгоритмов сортировки в модели разрешающих деревьев (раздел 9.1). (Указание. Не забудьте про равные элементы.)

28.2-5

Докажите, что для любого $i=1,2,\ldots,n-1$ *сортирующая сеть обязана содержать хотя бы один компаратор, соединяющий прямые* i *и* $i+1$.

28.3 Битонический сортировщик

Построение эффективной сортирующей сети мы начнём с так называемого битонического сортировщика, который сортирует так называемые битонические последовательности.

Мы называем битонической *(bitonic) любую последовательность, которая сначала возрастает, а потом убывает, или получается из такой циклическим сдвигом. Если записать элементы битонической последовательности по кругу, то минимальный и максимальный её элементы делят последовательность на два монотонных участка.*

Например, последовательности $\langle 1,4,6,8,3,2\rangle$, $\langle 6,9,4,2,3,5\rangle$ *и* $\langle 9,8,3,2,4,6\rangle$ *— битонические. Битонические последовательность нулей и единиц имеют вид либо* $1^i0^j1_k$, *либо* $0^i1^j0_k$, *где* $i,j,k \geqslant 0$. *Записанные по кругу, они состоят из двух групп — в одной нули, в другой единицы, и группы не смешиваются. Отметим, что монотонные последовательности являются частным случаем битонических.*

В этом разделе будет построен битонический сортировщик — сеть компараторов, правильно сортирующая битонические последовательности нулей и единиц. В упражнении 28.3-6 мы предложим вам показать, что он годится для произвольных битонических последовательностей.

Полуочиститель

Битонического сортировщик состоит из нескольких частей разных размеров, которые мы будеи называть "полуочистите-

Рисунок 28.7 Полуочиститель Half-Cleaner[8] и две различные входные последовательности нулей и единиц. Если входная последовательность является битонической последовательностью нулей и единиц, то после её обработки любой элемент верхней половины выхода меньше любого элемента нижней половины (или равен ему); одна из половин — битоническая, а другая состоит только из нулей или только из единиц (*"чистая"*).

Рисунок 28.8 Четыре возможности при работе сети Half-Cleaner[n]. На вход подаётся битоническая последовательность нулей и единиц; мы считаем, что она имеет вид 00...011...100...0. Нулевые участки на рисунке белые, единичные — серые. Входная последовательность разрезается на две половины, которые почленно сравниваются. (a)–(b) Случаи, когда точка раздела попадает в кусок из единиц. (c)–(d) Точка раздела проходит по нулевому участку. Во всех случаях выполнены утверждения (1)–(3) леммы 28.3.

лями" (half-cleaner). Полуочиститель размера n есть сеть глубины 1, в которой компараторы соединяюи провода одной половины с проводами другой (i и $i{+}n/2$ соединены при $i = 1, 2, \ldots, n/2$; предполагается, что n чётно). На рисунке 28.7 показан Half-Cleaner[8] *— полуочиститель размера 8.*

Основное свойство полуочистителя, объясняющее его название, таково:

Лемма 28.3.

Пусть на вход полуочистителю подана битоническая последовательность нулей и единиц. Получающаяся выходная последовательность обладает следующими свойствами: (1) ее верхняя и нижняя половины - битонические; (2) любой элемент верхней половины меньше любого элемента из нижней (или равен ему); (3) хотя бы одна из половин — чистая (и, следовательно, битоническая)

Доказательство. Будем предполагать, что вход имеет вид 00...011...100...0 *(случай* 11...100...011...1 *симметричен). Сеть* Half-Cleaner[n] *сравнивает вход a_i со входом $a_{i+n/2}$, поэтому возможны три расположения блока единиц относительно середины последовательности, причем случай, когда середина приходится на блок единиц, распадается на два подслучая. Как видно из рис. 28.8, лемма справедлива во всех четырёх случаях.*

Битонический сортировщик

Битонический сортировщик *(bitonic sorter) рекурсивно строится из полуочистителей, как показано на рис. 28.9. Сеть* Bitonic-Sorter[n] *состоит из полуочистителя* Half-Cleaner[n] *и двух экземпляров сети* Bitonic-Sorter[$n/2$]. *По лемме 28.3 полуочиститель делает из входной битонической последовательности две битонические последовательности (одна из них чистая) половинного размера. При этом любой элемент верхней меньше (или равен) любого элемента другой. После* *этого остаётся упорядочить каждую из них при помощи сети* Bitonic-Sorter$[n/2]$. *На рис. 28.9(a) показан общая структура сети, а на рис. 28.9(b) рекурсия развёрнута до конца. Таким образом строится битонический сортировщик с n входами, где n — произвольная степень двойки.*

Глубина $D(n)$ сети Bitonic-Sorter$[n]$ *даётся соотношением*

$$D(x) = \begin{cases} 0, & \text{если } n = 1, \\ D(n/2) + 1, & \text{если } n = 2^k\ k \geqslant 1. \end{cases}$$

из которого видно, что $D(n) = \lg n$.

Мы видим, что битонические последовательности нулей и единиц сортируются правильно, откуда следует (правило нуля и единицы для битонических последовательностей, упражнение 28.3-6), что и любые битонические последовательности сортируются правильно.

Упражнения

28.3-1

Сколько всего существует раличных битонических последовательностей нулей и единиц длины n?

28.3-2

Докажите, что сеть Bitonic-Sorter$[n]$ *содержит* $\Theta(n \lg n)$ *компараторов (напомним, что n есть степень двойки).*

28.3-3

Постройте битонический сортировщик с n входами глубины $O(\lg n)$ для случая, когда n не является степенью двойки.

28.3-4

Полуочиститель получает на вход произвольную битоническую последовательность. Докажите, что верхняя и нижняя половины выходной последовательности — битонические, а любой элемент верхней половины меньше любого элемента нижней (или равен ему).

28.3-5

Даны две последовательности нулей и единиц причем любой элемент первой меньше любого элемента второй или равен ему. Докажите, что хотя бы одна из последовательностей — чистая.

28.3-6

Докажите правило нуля и единицы для битонических последовательностей: если сеть компараторов упорядочивает все битонические последовательности нулей и единиц, то эта сеть упорядочивает любую битоническую последовательность.

Рисунок 28.9 Входная часть сети Merger[8] (a) отличается от сети Half-Cleaner[8] (b) тем, что нижние 4 провода перевёрнуты. (a) Входная часть сети Merger преобразует две возрастающие последовательности $\langle a_1, a_2, \ldots, a_{n/2}\rangle$ и $\langle a_{n/2+1}, a_{n/2+2}, \ldots, a_n\rangle$ в две битонические последовательности $\langle b_1, b_2, \ldots, b_{n/2}\rangle$ и $\langle b_{n/2+1}, b_{n/2+2}, \ldots, b_n\rangle$. (b) Тот же процесс в обозначениях полуочистителя: битоническая входная последовательность $\langle a_1, a_2, \ldots, a_{n/2}, a_{n/2+1}, \ldots, a_n\rangle$ преобразуется в две битонические последовательности $\langle b_1, b_2, \ldots, b_{n/2}\rangle$ и $\langle b_{n/2+1}, b_{n/2+2}, \ldots, b_n\rangle$.

28.4 Сливающая сеть

Сливающей сетью *(merging network) мы называем сеть компараторов, соединяющая две упорядоченые последовательности (половины входной последовательности) в одну упорядоченную последовательность. Такая сеть (мы назовём её* Merger[n]*) легко получается модификацией сети* Bitonic-Sorter[n].

Идея проста: для слияния двух упорядоченных последовательностей припишем (в обратном порядке) вторую последовательность к концу первой. При этом получится битоническая последовательность, которую уже можно упорядочить с помощью битонического сортировщика. Например, для объединения X = 00000111 *и* Y = 00001111 *мы приписываем к* X *перевёрнутую последовательность* Y^R = 11110000 *и получаем битоническую последовательность* XY^R = 0000011111110000. *Осталось применить к* XY^R *битонический сортировщик.*

Следуя этому плану, для построения сети Merger[n]*, сливающей последовательности* $\langle a_1, \ldots, a_{n/2}\rangle$ *и* $\langle a_{n/2+1}, \ldots, a_n\rangle$*, следует так перестроить первый полуочиститель сети* Bitonic-Sorter[n]*, чтобы добиться эффекта "переворачивания" второй последовательности. В обычном полуочистителе вход* a_i *сравнивается со входом* $a_{n/2+i}$*, (при* $i = 1, 2, \ldots, n/2$*), поэтому теперь мы будем* a_i *со входом* a_{n-i+1}*. На рис. 28.10 показан перестроенный полуочиститель в сравнении с обычным — разница в том, что провода в нижней его половине переставлены в обратном порядке. При этом верхняя и нижняя половины выходов по-прежнему обладают свойствами, указанными в лемме 28.3 (битоническая последовательность, записанная в обратном порядке, остаётся битонической).*

Для завершения слияния остаётся упорядочить обе половины при помощи двух копий сети Bitonic-Sorter[n]*. Полученая сеть* Merger[n] *показана на рис. 28.11. Её глубина такая же, как у сети* Bitonic-Sorter[n]*, то есть* $\lg n$.

Упражнения

28.4-1

Докажите следующий вариант правила нуля и единицы для сли-

Рисунок 28.10 Сеть Merger[n] соединяет две отсортированные последовательности в одну. Она получается из сети Bitonic-Sorter модификацией первого каскада: теперь сравниваются i-ые с начала и конца элементы. (a) Общая структура сети: первый каскад, затем два экземпляра сети Bitonic-Sorter[$n/2$]. (b) Полная схема сети (серые прямоугольники — структурные элементы, на рёбрах показаны значения для одного из возможных входов).

вающих сетей: если сеть компараторов правильно сливает любые две монотонные последовательности нулей и единиц, то она правильно сливает любые монотонные числовые последовательности.

28.4-2

Сколько тестовых последовательностей нулей и единиц необходимо, чтобы проверить. что сеть компараторов действительно является сливающей?

28.4-3

Докажите, что любая сеть компараторов, умеющая правильно сливать один элемент (a_1) с упорядоченной последовательностью длины $n-1$ ($a_2, a_3, \ldots, a_n$) в упорядоченную последовательность длины n, имеет глубину как минимум $\lg n$.

*28.4-4**

Пусть дана сливающая сеть со входами $a_1, \ldots, a_n$, которая сливает упорядоченные последовательности $\langle a_1, a_3, \ldots, a_{n-1}\rangle$ и $\langle a_2, a_4, \ldots, a_n\rangle$ (предполагаем, что n — степень двойки). Покажите, что такая сеть содержит $\Omega(n \lg n)$ компараторов. Чем интересна эта нижняя оценка?

(Указание. Разбейте множество компараторов на три части.)

[Что это за части? Какое значение вообще имеет порядок входов? Как решается этоа задача?]

*28.4-5**

Докажите, что любая сеть, сливающая две упорядоченные последовательности длины $n/2$ в одну (разрешается распределить входы между первой и второй сливаемыми подпоследовательностями произвольным образом) содержит $O(n \lg n)$ компараторов.

28.5 Сортирующая сеть

Теперь всё готово для построения сортирующей сети Sorter[n]*8, реализующей алгоритм сортировки слиянием из раздела 1.3.1. Устройство сети показано на рисунке 28.12.*

Сеть Sorter[n] *строится рекурсивно: две половины входной последовательности упорядочиваются сетями* Sorter[$n/2$]*, а затем соединяются при помощи сети* Merger[n]*. В качестве* Merger[2] *используется компаратор. Схема построения пока-*

Рисунок 28.11 Сеть Sorter[n] строится рекурсиво из сливающих сетей. (a) Рекурсивное описание сети Sorter[n]. (b) Рекурсия развёрнута. (c) Показано внутреннее строение сливающих сетей. Указаны глубины проводов и значения на них (для одного из возможных вариантов входов).

зана на рис. 28.12 (a), на рис. 28.12 (b) она развёрнута, а на рис. 28.12 (c) показано внутреннее устройство сливающих сетей.

Таким образом, сеть Sorter[n] *состоит из* $\lg n$ *каскадов. Первый из них содержит* $n/2$ *копий сети* Merger[2] *параллельно сливающих пары одноэлементных последовательностей. Второй уровень состоит из* $n/4$ *экземпляров сети* Merger[4], *которые сливают пары полученых на первом уровне двухэлементных последовательностей, и т. д. На* k*-ом уровне (при* $k = 1, 2, \ldots, \lg n$*) имеется* $n/2^k$ *экземпляров сети* Merger[2^k], *сливающих по две упорядоченные последовательности длины* 2^{k-1}. *Индуктивное построение гарантирует, что сеть правильно упорядочивает последовательности нулей и единиц. Из правила нуля и единицы (теорема 28.2) следует правильность работы сети на произвольных числовых последовательностях.*

Глубина $D(n)$ *сети* Sorter[n] *равна сумме глубин сетей* Sorter[$n/2$] *и* Merger[n] *(две копии сети* Sorter[$n/2$] *работают параллельно). Глубина сети* Merger[n] *равна* $\lg n$, *так что рекуррентная формула для* $D(n)$ *такова:*

$$D(x) = \begin{cases} 0, & \text{если } n = 1, \\ D(n/2) + \lg n, & \text{если } n = 2^k \text{ и } k \geqslant 1. \end{cases}$$

Из неё следует, что $D(n) = \Theta(\lg^2 n$.

Таким образом, можно сказать, что параллельный алгоритм сортировки может упорядочить n *чисел за время* $O(\lg^2 n)$, *используя сортирующие сети как модель параллельных вычислений.*

Упражнения

28.5-1

Сколько компараторов содержит сеть Sorter[n]*?*

28.5-2

Покажите, что глубина сети Sorter[n] *в точности равна* $\lg n(\lg n + 1)/2$. *(Число* n *есть степень двойки.)*

28.5-3

Рассмотрим компараторы нового типа, получающие на два своих входа упорядоченные последовательности длины k, *соединяющие их в одну последовательность и выдающие на верхний выход* k *меньших чисел, а на нижний –* k *больших чисел. (Таким образом, значениями на входах и выходах компараторов являются упорядоченные последовательности чисел длины* k*.) Покажите, что любая сортирующая сеть с* n *входами и выходами после замены компараторов на компараторы нового типа превращается* *в устройство, сливающее n упорядоченных последовательностей длины k в упорядоченную последовательности длины nk (разбитую на n частей длины k).*

28.5-4

Пусть даны $2n$ чисел $\langle a_1, \ldots, a_{2n}\rangle$ и требуется разделить их на n меньших и n больших чисел (внутри этих двух групп порядок может быть любым). Докажите, что это можно сделать с помощью сети глубины $O(1)$, если известно, что входная последовательность состоит из двух отсортированных половин $a_1, \ldots, a_n$ и $a_{n1+1}, \ldots, a_{2n}$.

*28.5-5**

Пусть дана сортирующая сеть с k входами глубины $S(k)$, а также сеть глубины $M(k)$, сливающая две упорядоченные последовательности длины k в одну длины $2k$. Постройте сеть глубины $S(k) + 2M(k)$, упорядочивающую все последовательности длины n, в которых каждый элемент отстоит не далее чем на k мест от своего правильного положения.

*28.5-6**

С матрицей размера $m \times m$ выполняют такие действия:

1. *Все нечетные строки упорядочивают по возрастанию.*
2. *Все нечетные строки упорядочивают по убыванию.*
3. *Упорядочить каждый столбец по возрастанию.*

Как будут отсортированы элементы матрицы после многократных повторений операции 1–3? Сколько повторений потребуется?

Задачи

28.1 Сортировка транспозициями.

Говорят, что сортирующая сеть использует лишь транспозиции *(is a transposition network), если каждый её компаратор соединяет две соседние прямые (рис. 28.3)*

a. *Покажите, что любая такая сеть содержит $\Omega(n^2)$ компараторов.*

b. *Чтобы убедиться в правильности работы такой сети, достаточно проверить, что она правильноо сортирует последовательность $\langle n, n-1, \ldots, 1\rangle$. (Указание: Индуктивное рассуждение следует доказательству леммы 28.1)*

Чётно-нечётная сортирующая сеть *(odd-even sorting network) для 8 входов показана на рис. 28.13. В общем случае она содержит n уровней компараторов: для $i = 2, 3, \ldots n-1$ и $d = 1, 2, \ldots, n$ прямая номер i на глубине d соединена компаратором с прямой номер $i + (-1)^{i+d}$*

c. Докажите, что действительно получается сортирующая сеть.

28.2 Четно-нечетное слияние по Бэтчеру

В разделе 28.4 сливающая сеть строилась на основе битонического сортировщика. В этой задаче рассматривается другой способ её построения — чётно-нечетная сливающая сеть *(odd-even merging network). Будем считать, что n — степень двойки, и опишем сеть, сливающую две упорядоченые последовательности $\langle a_1, \ldots, a_n\rangle$ и $\langle a_{n+1}, \ldots, a_{2n}\rangle$ в одну. Строится она рекурсивно. Возьмём две сети половинного размера и используем их для параллельного слияния двух пар последовательностей длины $n/2$: последовательность $\langle a_1, a_3 \ldots, a_{n-1}\rangle$ сливается с $\langle a_{n+1}, a_{n+3} \ldots, a_{2n-1}\rangle$, а $\langle a_2, a_4 \ldots, a_n\rangle$ — с $\langle a_{n+2}, a_{n+4} \ldots, a_{2n}\rangle$. На последнем этапе стоят n компараторов, i-ый из которых сравнивает число на прямой $2i - 1$ с числом на прямой $2i$.*

a. *Нарисуйте такую сеть для $n = 4$.*

b. *Используя правило нуля и единицы, докажите правильность работы построенной сети для произвольного n, являющегося степенью двойки.*

с *Каковы размеры и глубина чётно-нечётной сливающей сети с $2n$ входами?*

28.3 Сети коммутаторов

Сеть коммутаторов *(permutation network) с n входами и n выходами состоит из проводов и переключателей, позволяющих соединять входы с выходами всеми возможными $n!$ способами. На рис. 28.14 (a) показана сеть коммутаторов P_2. Она состоит из единственного коммутатора, который имеет два положение: прямое и перекрёстное.*

a. *Покажите, что если в любой сортирующей сети компараторы заменить на коммутаторы, то получается сеть, реализующая все перестановки: для любой перестановки π можно установить коммутаторы в такие положения, что вход i соединён с выходом $\pi(i)$, при всех $i = 1, 2 \ldots, n$.*

На рис. 28.14 (b), показана сеть коммутаторов P_8 на 8 позиций, составленная из двух сетей P_4 и восьми отдельных коммутаторов. На рисунке сеть реализует перестановку $\pi = \langle 4, 7, 3, 5, 1, 6, 8, 2\rangle$, при этом верхняя сеть P_4 реализует перестановку $\langle 4, 2, 3, 1\rangle$, а нижняя — $\langle 2, 3, 1, 4\rangle$.

b. *В какое состояние надо перевести коммутаторы и какие перестановки установить в верхней и нижней P_4-сетях, чтобы сеть P_8 реализовала перестановку $\langle 5, 3, 4, 6, 1, 8, 2, 7\rangle$?*

Пусть n — степень числа 2. Рассмотрим сеть P_n, построенную из двух сетей $P_{n/2}$ и отдельных коммутаторов аналогично сети P_8.

c. *Докажите, что сеть P_n способна реализовать любую перестановку и укажите алгоритм, который за время $O(n)$ (время измеряется обычным образом, для последовательной машины с произвольным доступом) указывает положения коммутаторов и перестановки для подсетей, реализуюжие заданную перестановку* *для сети P_n.*

d. *Каковы глубина и размер сети P_n? Найдите общее время, за которое построенный в пункте* **b** *алгоритм рассчитает положения коммутаторов, включая коммутаторы для подсетей нижних уровней.*

e. *Покажите, что любая сеть коммутаторов с n входами, реализующая все перестановки, неизбежно реализует какую-то перестановку по крайней мере двумя способами.*

Замечения

Кнут[123] обсуждает свойства сортирующих сетей и их историю. Видимо, их впервые рассмотрели Армстронг (P.N. Armstrong), Нельсон (R.J. Nelson) и Коннор (D.J.O'Connor) в 1954 году. В начале 1960-х годов Бэтчер (K.E. Batcher) придумал сеть, сливающую две числовые последовательности длины n за время $O(\lg n)$, используя чётно-нечётное слиние (задача 28.2). Он также показал, как с помощью таких сетей упорядочить n чисел за время $O(\lg^2 n)$. Чуть позже Бэтчер придумал битонический сортировщик глубины $O(\lg n)$, аналогичную описанному в разделе 28.3. Согласно Кнуту, правило нуля и единицы Кнут доказал Бурицій (W.G. Bouricius, 1954) в контексте разрешающих деревьев.

Долгое время оставался открытым вопрос о существрвании сортирующей сети глубины $O(\lg n)$. В 1983 году на него был дан положительный ответ: Айтай, Комлос и Семереди [8] построили сортирующую сеть для n чисел глубины $O(\lg n)$ из $O(n \lg n)$ компараторов. К сожалению, константы в этой $O(n)$-оценке (многие тысячи, если не больше) препятствуют практическому использованию этой сети.

29 Арифметические схемы

До сих пор, говоря о времени работы алгоритма, мы считали, что арифметические операции (сложение, вычитание, умножение и деление) выполняются за постоянное время, не зависящее от размера чисел. Такое приближение разумно с точки зрения практики, поскольку для чисел, помещающихся в разрядную сетку компьютера, эти времена не так уж сильно различаются. Но при алгоритмической реализации этих операций мы видим, что время работы зависит от размера входов. Скажем, школьный алгоритм сложения двух n-значных чисел в десятичной системе требует $\Theta(n)$ *элементарных действий.*

В этой главе рассматриваются схемы для арифметических действий. При последовательной реализации любое действие с двумя n-разрядными числами требует $\Omega(n)$ *битовых операций (за меньшее время мы даже не прочтём входы). Уменьшения времени можно достичь при использовании схем, элементы которых работают параллельно. В этой главе мы построим такие схемы для сложения и умножения (вычитание аналогично сложению; по поводу деления см. задачу 29.1). Предполагается, что на входы поступают n-значные числа в двоичной системе.*

Мы начнём в разделе 29.1 с определения и примеров схем из функциональных элементов; время работы будет определяться глубиной схемы. Нашим первым примером будет сумматор — базовый элемент для конструкций этой главы. Раздел 29.2 посвящен двум схемам для сложения: схеме каскадного сложения (время работы $\Theta(n)$*) и схеме сложения с предвычислением переносов, (время* $O(\lg n)$*). Там же рассмотрена схема сложения с запоминанием переносов, позволяющая свести (за время* $\Theta(1)$*) сложение трёх чисел к сложению двух. В разделе 29.3 строятся две схемы для умножения: матричный умножитель (время работы* $\Theta(n)$*) и умножитель с помощью дерева Wallace'а (время* $\Theta(\lg n)$*). Наконец, в разделе 29.4 рассказано о схемах с элементами задержки, которые позволяют многократно использовать одни и те же функциональные элементы (и тем самым уменьшить общее количество элементов в схеме).*

29.1 Схемы из функциональных элементов

Как и сортирующие сети в главе 28, схемы из функциональных элементов являются параллельной моделью вычислений: несколько элементов схемы могут работать одновременно. В этом разделе мы дадим определение схемы из функциональных элементов и приведём примеры.

29.1.1 Функциональные элементы

Арифметические схемы строятся из функциональных элементов, соединенных проводниками. Функциональный элемент (combinational element) имеет входы и выходы; его выходной сигнал является функцией входных. Если входные и выходные значения являются нулями и единицами, элемент называется логическим (boolean combinational element, logic gate); как обычно, 0 *обозначает "ложь", а* 1 *— "истину" .*

Четыре основных логических элемента, используемые в этой главе, показаны на рис. 29.1 : NOT (отрицание), AND (логическое И), OR (логическое ИЛИ) и XOR (исключающее ИЛИ); ещё два — NAND и NOR — используются в некоторых упражнениях). Элемент NOT имеет один вход x, на который подается 0 *или* 1, *и один выход z, значение на котором противоположно значению на входе. Остальные три элемента имеют по два входа (x и y) и по одному выходу (z).*

Рисунок 29.1 29.1 Шесть основных логических элементов и соответствующие таблицы. (a) Элемент NOT. (b) Элемент AND. (c) Элемент OR. (d) Элемент XOR. (e) Элемент NAND (Not-AND). (e) Элемент NOR (Not-OR).

Работа элемента (как и любой схемы, составленной из них) может быть описана таблицей, которая указывает выходные значения для всех наборов значений на входах. Например, из таблицы для XOR видно, что входы $x = 0$ и $y = 1$ дают на выходе $z = 1$. Для обозначения операции NOT традиционно используется символ $\neg$, для AND - символ $\wedge$, для OR - символ $\vee$, а для XOR - символ $\oplus$. Например, $0 \oplus 1 = 1$.

Реальные функциональные элементы работают не мгновенно. Правильное значение на выходе появляется лишь через некоторое время после того, как значения на входах установятся. Это время называется задержкой (propagation delay). Мы предполагаем, что задержка одна и та же для всех элементов.

29.1.2 Схемы из функциональных элементов

Мы рассматриваем схемы из функциональных элементов (combinational circuits), соединённых проводами (wires). При этом не должно образовываться циклов (см. ниже). Провод соединяет выход одного элемента со входом другого; выходное значения первого элемента используется вторым как входное. Провод должен быть подключен не более чем к к одному выходу, но может соединять его сразу с несколькими входами. Число входов элементов, подсоединенных к данному проводу, называется его выходной степенью (fan-out). Если провод не подсоединен к выходам элементов, то он является входом схемы, получая значение извне. Если провод не подключен к входам элементов, то он является выходом схемы и выдаёт наружу результат работы схемы (внутренний провод также может, разветвляясь, служить выходом схемы). Схемы не содержат циклов (цикл появляется, если выход элемента является входом второго, выход второго – входом третьего, ..., наконец, выход n-го – входом первого). Они также не содержат элементов задержки (если это не оговорено специально — см. раздел 29.4).

29.1.3 Сумматор

На рис. 29.2 изображен сумматор (full adder), на входы x, y и z которого поступают три бита. Значения на выходах c и s даются таблицей

x	y	z	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Другими словами, s есть функция чётности (parity) входных аргументов:

$$s = parity(x, y, z) = x \oplus y \oplus z \tag{29.1}$$

а c – функция большинства (majority):

$$c = majority(x, y, z) = (x \wedge y) \vee (y \wedge z) \vee (x \wedge z) \tag{29.2}$$

Функции чётности и большинства имеют смысл для любого числа аргументов: функция "чётность" равна 1, когда среди её ар-

гументов нечётное число единиц, а функция " большинство" — когда более половины её аргументов равны 1.

Заметьте, что в совокупности биты c и s дают двоичную запись суммы $x + y + z$. *Например, если* $x = 1$, $y = 0$ *и* $z = 1$, *то* $\langle c, s\rangle = \langle 10\rangle$, *а* $x + y + z = 2 = 10_2$.

Рисунок 29.2 Сумматор. (a) В момент 0 биты поступают на вход. (b) Через единицу времени появляются значения на выходах элементов A-D глубины 1. (c) Ещё через единицу времни появляются значения на выходах элементов E и F глубины 2. (d) Ещё через единицу времени (момент времени 3) на выходе элемента G появляется результат работы всей схемы.

Элементы A *и* E *можно заменить на один элемент XOR с тремя входами а элементы* F *и* G *— на один элемент OR с тремя входами. Число входов элемента называют его входной степенью (fan-in).*

Будем считать, что каждый элемент имеет единичную задержку, а входные значения поступают в момент 0 *(рис. 29.2, a). С этого момента входы элементов* A-D *(и только их) стабильны в этот момент, поэтому с момента* 1 *их выходные значения стабильны (рис. 29.2, b). Тем самым элементы* A–D *работают параллельно. С момента* 1 *входные значения у* E *и* F *(но не у* G*!) стабильны, поэтому с момента* 2 *стабильны их выходные знаения (рис. 29.2, c), в том числе выходной бит* s. *Однако бит* c *ещё не получен — элемент* G *имеет стабильные входы с момента* 2 *и выдаёт* c *в момент 3 (рис. 29.2, d).*

29.1.4 Глубина схемы

Как и в случае сортирующих сетей в главе 28, задержка схемы определяется наибольшим количеством элементов на путях от входов схемы к выходам. Эта характеристика схемы называется её глубиной (depth) и определяет время её работы. По определению входы схемы имеют глубину 0, *а выходные провода элемента со входами глубины* $d_1, \ldots, d_n$ *имеют глубину* $\max\{d_1, \ldots, d_n\} + 1$. *Поскольку в схеме нет циклов, это индуктивное определение корректно. Глубиной элемента называется глубина его выходов, а глубиной схемы - наибольшая глубина составляющих её элементов.*

Если все функциональные элементы производят действие за одно и то же время t, *то время работы схемы глубины* d *не превосходит* dt. *На рис. 29.2 указана глубина всех элементов сумматора. Поскольку наибольшую глубину (3) имеет элемент* G, *глубина схемы также равна* 3.

Иногда схема может работать несколько быстрее. Например, если на один из входов элемента AND подан 0, *то на выходе по-*

явится 0, *даже если второе входное значение будет всё ещё меняться. Однако на практике время работы схемы чаще всего пропорционально глубине.*

29.1.5 Размер схемы

При разработке схемы, реализующей данную функцию, стараются уменьшить не только её глубину, но и размер *(size) — число функциональных элементов. Например, описанный выше сумматор имеет размер* 7.

Пусть задана некоторая функция, и мы хотим реализовать её схемой наименьшего размера. Чтобы этот вопрос имел смысл, надо договориться, какие элементы разрешено использовать — в противном случае можно считать, что схема состоит из единственного элемента, реализующего требуемую функцию. Такиим образом, минимальный размер схемы зависит от набора разрешённых элементов.

Однако если нас интересует не одна схема, а семейтво однотипных схем (например, мы строим сумматор n-разрядных чисел для любого n), увеличение набора разрешённых элементов (если он остаётся конечным) уменьшает размер схемы всего лишь в константу раз по сравнению с базовым набором (AND, OR, NOT), поскольку все разрешённые элементы можно составить из базовых.

29.1.6 Упражнения

29.1-1. *Как изменятся значения на рис. 29.2 (a)-(d), если* $x = 1$, $y = 1$, $z = 1$*?*

29.1-2. *Постройте из* $n - 1$ *элемента XOR схему для функции "чётность" с* n *входами, глубина которой не превосходит* $\lceil \lg n \rceil$.

29.1-3. *Докажите, что для любой таблицы значений соответствующий логический элемент можно реализовать схемой из элементов AND, OR и NOT.*

29.1-4. *Докажите, что в предыдущей задаче можно обойтись только элементами типа NAND.*

29.1-5. *Постройте схему для функции XOR из четырёх элементов NAND.*

29.1-6. *Пусть схема* C *имеет* n *входов,* n *выходов и глубину* d. *Соединим два экземпляра* C*, присоединив выходы первого ко входам второго. Какова может быть глубина такой схемы? (Укажите наибольшее и наименьшее возможные значения.)*

29.2 Схемы для сложения

В этом разделе приведены три схемы для сложения n-битовых двоичных чисел. Каскадное сложение (ripple-carry addition) позволяет сложить два n-значных числа за время $\Theta(n)$ с помощью схемы размера $\Theta(n)$. Время можно уменьшить до $O(\lg n)$, используя сложение с предвычислением переносов (carry-lookahead addition), причем соответствующая схема также имеет размер $\Theta(n)$. Наконец, сложение с запоминанием переносов (carry-save addition) за время $O(1)$ сводит сложение трёх n-разрядных чисел к сложению n-разрядного и $(n+1)$-разрядного чисел. Схема также имеет размер $\Theta(n)$.

29.2.1 Каскадное сложение

Неотрицательное целое число a записывается в двоичной системе как последовательностью n битов $\langle a_{n-1}, a_{n-2}, \ldots, a_0\rangle$, причем $n \geqslant \lceil \lg(a+1)\rceil$ и

$$a = \sum_{i=0}^{n-1} a_i 2^i.$$

При сложении по двум n-значным числам $a = \langle a_{n-1}, a_{n-2}, \ldots, a_0\rangle$ и $b = \langle b_{n-1}, b_{n-2}, \ldots, b_0\rangle$ строится $(n+1)$-значное число $s = \langle s_n, s_{n-2}, \ldots, s_0\rangle$, равное их сумме (пример на рис. 29.3).

Рисунок 29.3 29.3 При сложении 8-значных чисел $a = \langle 01011110\rangle$ и $b = \langle 11010101\rangle$ получается 9-значное число $s = \langle 100110011\rangle$. В i-м столбце указаны i-е разряды чисел a, b и s, а также бит переноса c_i. Входной бит переноса c_0 всегда равен 0.

При сложении столбиком (справа налево) мы складываем в i-м разряде a_i, b_i и входной бит переноса (carry-in bit) c_i. Младший разряд суммы записывается в i-ый разряд ответа (s_i), а старший становится выходным битом переноса (carry-out bit) c_{i+1} и используется при сложении в следующем разряде. В младший разряд ничего не переносится, поэтому $c_0 = 0$. Последний перенос c_n становится старшим разрядом суммы s_n. Поскольку $s_i = parity(a_i, b_i, c_i)$, а $c_{i+1} = majority(a_i, b_i, c_i)$, для каждого шага можно годится описанный выше сумматор.

Таким образом, n-разрядный каскадный сумматор состоит из последовательно соединённых простых сумматоров $FA_0, FA_1, \ldots, FA_{n-1}$, так что выход c_{i+1} сумматора FA_i является входом для FA_{i+1}. На входе c_0 фиксировано значение 0, не зависящее от входов (см. рис. 29.4).

Рисунок 29.4 Каскадный сумматор для $n = 8$. Ромбик справа означает, что значение на входе фиксировано.

Поскольку бит переноса проходит через все сумматоры, глубина каскадного сумматора равна n *(а глубина элемента* FA_i *равна* $i + 1$*). Поэтому время работы составляет* $\Theta(n)$*. Размер схемы равен* $\Theta(n)$*.*

29.2.2 Сложение с предвычислением переносов

В каскадном сумматоре бит переноса c_i *вычисляется в момент времени* i*. Значения* a_i *и* b_i*, однако, известны с самого начала. В некоторых случаях они определяют бит переноса* c_i*:*

- *если* $a_i = b_i = 0$*, то* $c_i = 0$ *(перенос "поглощается" (kill))*
- *если* $a_i = b_i = 1$*, то* $c_i = 1$ *(перенос "порождается" (generate))*

Однако если один из битов a_i *и* b_i *равен 1, а другой 0, но* c_{i-1} *существен; именно,*

- *если* $a_i \neq b_i$*, то* $c_i = c_{i-1}$ *(перенос распространяется (propagate))*

Каждому разряду, следовательно, соответствует один из трёх типов переноса *(carry statuses): k, (kill), g (generate) или p (propagate) (см. рис. 29.5). Этот тип известен заранее, что позволяет уменьшить время работы схемы сложения.*

Рисунок 29.5 Выходной бит переноса и тип переноса сумматора FA_{i-1} в зависимости от a_i и b_i.

Зная типы переноса для соседних сумматоров (($i-1$*)-го и* i*-го), можно определить тип переноса для их соединения, считая* c_{i-1} *входным битом, а* $_{i+1}$ *— выходным: зная, что случается с битом переноса на каждом шаге, можно рассчитать, что произойдёт за два шага, то есть как зависит* c_{i+1} *от* c_{i-1}*. Если* i*-й разряд имеет тип k или g, то соединение имеет тот же тип переноса. Если же* i*-й разряд имеет тип переноса p, то тип переноса для соединения совпадает с типом (*$i-1$*)-го разряда (см. рис. 29.6).*

Рисунок 29.6 Тип переноса соединения двух сумматоров (таблица операции $\otimes$).

Таблицу на рис. 29.6 можно рассматривать как определение операции (композиции типов переноса) на множестве $\{k, g, p\}$ *; она будет обозначаться символом* $\otimes$*. Эта операция ассоциативна (см. упражнение 29.2-2).*

С помощью этой операции тип переноса для некоторого участка числа выражается через типы переносов отдельных рязрядов. Обозначим через x_i тип переноса в i-м разряде.

$$x_i = \begin{cases} k, if a_{i-1} = b_{i-1} = 0; \\ g, if a_{i-1} = b_{i-1} = 1; \\ p, if a_{i-1} \neq b_{i-1}. \end{cases} \qquad (29.3)$$

Тогда зависимость, скажем, бита c_7 от c_4 определяется композицией $x_5 \otimes x_6 \otimes x_7$.

Поскольку в нулевой разряд переноса от младших разрядов не поступает, условно положим $x_0 = k$. Тогда перенос на выходе $i -$ разряда определяется композицией $x_0 \otimes x_1 \otimes \ldots \otimes x_i$: $c_i = 0$, если композиция равна k, и $c_i = 1$, если композиция равна g. (Значение p для композиции невозможно, поскольку для этого все члены должны быть равна p, а это не так для x_0.)

Более формально это записывается так. Положим $y_0 = k$ и определим $y_1, y_2, \ldots y_n$ так:

$$y_i = x_i \otimes y_{i-1} = x_0 \otimes x_1 \otimes \ldots \otimes x_i \qquad (29.3)$$

Другими словами, $y_0, \ldots y_n$ называются префиксами (*prefixes*) выражения $x_0 \otimes x_1 \otimes \ldots \otimes x_n$. (Общая задача о параллельном вычислении префиксов рассмотрена в гл. 30). На рис. 29.7 показаны значения x_i и y_i для примера рис. 29.3.

Рисунок 29.7 29.7. Значения x_i и y_i для примера рис. 29.3.

Теперь сказанное выше запишется так:

Лемма 29.1. При всех $i = 0, 1, 2, \ldots, n$:

1. Если $y_i = k$, то $c_i = 0$.
2. Если $y_i = g$, то $c_i = 1$
3. Случай $y_i = p$ невозможен.

Доказательство. Индукция по i. При $i = 0$ по определению $y_0 = x_0 = k$ и $c_0 = 0$. Пусть утверждение леммы выполнено для $i - 1$. Возможны три случая.

1. Если $y_i = k$, то либо $x_i = k$ (и тогда $_i = 0$), либо $x_i = p$ и $x_{i-1} = k$. Тогда по предположению индукции $c_{i-1} = 0$, а $c_i = c_{i-1}$ (бит переноса сохраняется), поэтому и в этом случае $c_i = 0$.

2. Случай $y_i = g$ аналогичен.

3. Если $y_i = p$, то обязательно $x_i = p$ и $y_{i-1} = p$, но последнее равенство невозможно по предположению индукции.

Лемма доказана.

Таким образом, вычисление битов переноса c_i сводится к вычислению префиксов y_i. Оставшиеся действия выполняются за время $\Theta(1)$ -достаточно подать биты переноса на входы сумматоров.

29.2.3 Вычисление типов переноса с помощью параллельной префиксной схемы

Здесь мы рассмотрим схему, использующую возможность параллельных вычислений и позволяющую за время $O(\lg n)$ вычислить все типы переноса y_i. Она имеет размер $\Theta(n)$.

Введём некоторые обозначения. При $0 \leqslant i \leqslant j \leqslant n$ положим

$$[i, j] = x_i \otimes x_{i+1} \otimes \ldots \otimes x_j$$

В частности, $[i, i] = x_i$. Если $0 \leqslant i < j \leqslant k \leqslant n$, то

$$[i, k] = [i, j-1] \otimes [j, k] \tag{29.4}$$

поскольку операция композиции $\otimes$ ассоциативна. Наша цель — вычислить все $y_i = [0, i]$.

Схема состоит из одинаковых элементов, каждый из которых вычисляет композицию $\otimes$. Идея её проста: на первом уровне параллельно вычисляются композиции пар ($[1, 2], [3, 4], \ldots$), затем четвёрок и так далее, пока мы не дойдем до композиции всех элементов. Затем мы движемся в обратном направлении, пока не доходим до искомых y_i. Фрагмент дерева (внутренняя вершина и её потомки) показан на рис. 29.8. На рис. 29.9 показана полная схема для $n = 8$. Входы $x_1, \ldots x_n$ и выходы $y_0, \ldots, y_{n-1}$ расположены в листьях дерева, а вход x_0 и выход y_n - в корне, так что данные движутся по дереву сначала от листьев к корню, а потом обратно.

Рисунок 29.8 Схема для параллельного вычисления префиксов. Показана внутренняя вершина, отвечающая за $x_i \ldots x_k$. Левое поддерево объединяет входы $x_i, \ldots, x_{j-1}$, правое — входы $x_j, \ldots, x_k$. Два элемента $\otimes$ (один используется при прямом ходе, другой — при обратном) вычисляют $[i, k] = [i, j-1] \otimes [j, k]$ и $[0, j-1] = [0, i-1] \otimes [i, j-1]$.

Рисунок 29.9 29.9 Случай $n = 8$. (a) Общая структура и вычисляемые значения (b) Значения, соответствующие примеру на рис. 29.3

Два элемента $\otimes$ в каждом узле работают в разное время (имеют разную глубину): левый (на рис. 29.8) работает "на пути вверх", а правый — на пути вниз. Убедиться в том, что схема работает правильно, можно по индукции. Предполагая, что поддеревья на рис. 28.8 вычисляют $[i, j-1]$ и $[j, k]$, мы получаем, что левый элемент $\otimes$ вычисляет $[i, k]$так что вычисления снизу вверх правильны. Проследим за движением информации вниз. Предполагая, что в вершину на рис. 29.8 сверху приходит правильное значение $[0, i-1]$, мы видим, что правый элемент $\otimes$ правильно

Рисунок 29.10 29.10 Сумматор с предвычислением переносов (показан случай $n = 8$) состоит из $n + 1$ блока KPG с номерами от 0 до n. Блок KPG_i получает на вход a_i и b_i, вычисляет тип переноса x_i и обрабатывает выходное значение y_i, выдавая i-ый бит суммы s_i. Показаны значения для примера рис. 29.3

вычисляет $[0, j-1] = [0, i-1] \otimes [i, j-1]$. *Это значение передаётся правому сыну; левому передаётся неизменённое значение* $[0, i-1]$.

Строение корня дерева (дополнительный элемент $\otimes$*) показано на рис. 29.9.*

Если n *является степенью двойки, то в параллельной префиксной схеме* $2n-1$ *элемент. Время работы составляет* $O(\lg n)$*, поскольку дерево имеет высоту* $\lg n$*, а данные проходят по нему дважды (вверх и вниз).*

29.2.4 Сумматор с предвычислением переносов: окончание

Полная конструкция n*-разрядного* сумматора с предвычислением переносов *(carry-lookahead adder) показана на рис. 29.10. Кроме разобранной схемы параллельного вычисления префиксов, него входит* $n+1$ *блок KPG. Блок KPG с индексом* i *вычисляет по входам* a_i, b_i *тип переноса* x_i *и передаёт его наверх, а затем, получив сверху значение* y_i *(которое, согласно лемме 29.1, соответствует биту переноса* c_i*), вычисляет с помощью сумматора* FA_i *значение* i*-го бита суммы* s_i*. Значения* $a_{n+1} = 0$, $b_{n+1} = 0$ *и* $x_0 = \mathbf{k}$ *зафиксированы. Поскольку все операции, кроме выполняемых параллельной префиксной схемой, требуют времени* $O(1)$*, общее время работы схемы составляет* $O(\lg n)$*. Размер схемы равен* $\Theta(n)$.

29.2.5 Сложение с запоминанием переносов

Как ни странно, сложение трёх чисел почти не требует дополнительных затрат по сравнению со сложением двух: глубина увеличивается всего на несколько единиц. Пусть $x = \langle x_{n-1}, x_{n-2}, \ldots, x_0 \rangle$, $y = \langle y_{n-1}, y_{n-2}, \ldots, y_0 \rangle$ *и* $z = \langle z_{n-1}, z_{n-2}, \ldots, z_0 \rangle$ *— три* n*-разрядных числа. Схема* сложения с запоминанием переносов *(carry-save adder) находит два числа* $u = \langle u_{n-2}, u_{n-2}, \ldots, x_0 \rangle$ *(*n *битов) и* $v = \langle v_n, v_{n-1}, \ldots, v_0 \rangle$ *(*$n+1$ *битов), для которых*

$$u + v = x + y + z.$$

Она делает это следующим образом (рис. 29.11 (b)):

$$u_i = parity(x_i, y_i, z_i),$$

$$v_{i+1} = majority(x_i, y_i, z_i)$$

(для $iu = 0, 1, \dots, n-1$*; бит* v_0 *всегда равен* 0*). Рассмотрим числа* $u = \langle u_{n-1}, u_{n-2}, \dots, u_0 \rangle$ *и* $v = \langle v_n, v_{n-1}, \dots, v_0 \rangle$*. Легко видеть, что*

$$x + y + z = u + v$$

Числа u *и* v *могут быть вычислены за время* $O(1)$ *с помощью* n *сумматоров* $FA_0, \dots, FA_{n-1}$ *(рис. 29.11). Для сложения чисел* u *и* v *используется сумматор с предвычислением переносов. Это требует времени* $O(\lg n)$*, всего получается* $O(1)+O(\lg n)$*, то есть* $O(\lg n)$*. Хотя та же асимптотическая оценка получается при использовании двух сумматоров с предвычислением переносов, но на практике разница (примерно в два раза) существенна.*

Аналогичный приём (сведение сложения трёх чисел к сложению двух) играет важную роль в быстрых схемах для умножения (см. раздел 29.3).

Рисунок 29.11 (a). Схема сложения с запоминанием переносов. Для сложения трёх n-битовых чисел x, y, z вычисляются числа u (n битов) и v ($n+1$ битов), для которых $x+y+z = u+v$. (b) 8-разрядный сумматор с запоминанием переносов. Каждый из сумматоров FA_i получает на вход x_i, y_i, z_i и выдаёт бит суммы u_i и бит переноса v_i. Мы полагаем $v_0 = 0$.

Упражнения

29.2-1 Пусть $n = 8$*,* $a = \langle 01111111 \rangle$*,* $b = \langle 00000001 \rangle$*. Укажите биты переноса и типы переноса во всех разрядах, а также значения на всех проводах схемы рис. 29.9, включая выходы* $y_0, y_1, \dots, y_8$*.*

29.2-2 Докажите, что операция $\otimes$ *(заданная таблицей рис. 29.5) ассоциативна.*

29.2-3 Объясните, как должна быть устроена параллельная префиксная схема, если n *не является степенью двойки (например, рассмотрите случай* $n = 11$*). Каково время работы такой схемы?*

29.2-4 Покажите внутреннее устройство схемы KPG, если значения на входах и выходах кодируются так: на входах $\langle 00 \rangle$ *соответствует* `k`*,* $\langle 11 \rangle$ *–* `g`*,* $\langle 01 \rangle$ *или* $\langle 10 \rangle$ *–* `p`*; на выходах* 0 *соответствует* `k`*,* 1 *–* `g`*.*

29.2-5 Укажите глубину каждого провода на рис. 29.9 (a). Найдите критический *(самый длинный) путь (critical path) от входов к выходам и покажите, что его длина есть* $O(\lg n)$*. Найдите узел, два элемента* $\otimes$ *которого срабатывают с интервалом* $\Theta(\lg n)$*. Есть ли узлы, в котором два элемента* $\otimes$ *срабатывают одновременно?*

29.2-6 Следую образцу рис. 29.12, укажите способ построения схемы вычисления префиксов для любого числа входов, являющегося степенью двойки. Докажите, что схема имеет глубину $\Theta(\lg n)$ *и размер* $\Theta(n \lg n)$*. Докажите, что схема работает правильно, разбив её на блоки и рассуждая индуктивно.*

Рисунок 29.12 29.12 Параллельная префиксная схема для упражнения 29.2-6

29.2-7 Какова максимальная выходная степень каждого провода в схеме с предвычислением переносов? Постройте схему для сложения глубины $O(\lg n)$ *и размера* $\Theta(n)$*, в которой все провода имеют выходную степень* $O(1)$*.*

29.2-8 Постройте счётчик единиц *— схему с* n *входами и* $\lceil \lg(n+1) \rceil$ *выходами, выдающую число единиц среди входов, записанное в двоичной системе. (Например, вход* $\langle 10011110 \rangle$ *порождает выход* $\langle 101 \rangle$ *(пять единиц). Глубина схемы должна составлять* $O(\lg n)$*, размер —* $\Theta(n)$*.*

29.2-9 Постройте схему для сложения глубины* 4 *и полиномиально зависящего от* n *размера, используя элементы AND и OR с произвольным числом входов. (Дополнительный вопрос: сделайте то же самое для глубины* 3*.)*

29.2-10 Пусть на каждом входе каскадного суматора* n*-разрядных чисел с равной вероятностью появляются значения* 0 *и* 1*, причём разные входы независимы. Докажите, что с вероятностью не менее* $1 - 1/n$ *никакой перенос не распространяется далее чем на* $O(\lg n)$ *разрядов.*

29.3 Схемы для умножения

Наиболее простой алгоритм умножения "столбиком" состоит в сложении "сдвигов" одного из сомножителей: $2n$*-битовое произведение двух* n*-битовых чисел* $a = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$ *и* $b = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$ *может быть записано как*

$$p = a \cdot b = \sum_{i=0}^{n-1} m^{(i)},$$

где $m^{(i)} = a \cdot b_i \cdot 2^i$ *получается из* a *сдвигом всех разрядов на* i *единиц влево (и заполнением освободившихся разрядов нулями), если* $b_i = 1$*, и равно* 0*, если* $b_i = 0$*) (рис. 29.13). Числа* m_i *называются* частичными произведениями*; каждое ненулевое частичное произведение соответствет ненулевому разряду в* b*.*

Рисунок 29.13 29.13 Умножение чисел $a = \langle 1110 \rangle$ и $b = \langle 1101 \rangle$ столбиком; произведение $p = a \cdot b = \langle 10110110 \rangle$. Каждое частичное произведение $m^{(i)}$ получается из a сдвигом на i разрядов влево, если $b_i = 1$. В противном случае оно равно нулю.

В этом разделе рассматриваются две схемы для умножения. Матричный умножитель работает за время $\Theta(n)$ *и имеет раз-*

мер $\Theta(n^2)$. Дерево Уоллеса (Wallace-tree multiplier) также имеет размер $\Theta(n^2)$, но работает быстрее, за время $\Theta(\lg n)$. Обе схемы используют умножение столбиком.

29.3.1 Матричный умножитель

Матричный умножитель работает в три этапа: (1) вычисляет частичные произведения; (2) складывает их, используя сложение с запоминанием переносов, пока не остается два числа; (3) складывает эти два числа (с помощью каскадного сумматора или с предвычислением переносов).

Матричный умножитель *(array multiplier) показан на рис. 29.14; входные биты a_i соответствуют вертикальным проводам, биты b_i — горизонтальным. Каждый входной бит поступает на входы n элементов AND, которые вычисляют разряды частичных произведений:*

$$m_{j+i}^{(i)} = a_j \cdot b_i = a_j \, AND \, b_i$$

Число элементов AND равно n^2; все они работают одновременно, вычисляя биты частичных произведений (кроме тех, что заведомо равны 0). Затем сумматоры с запоминанием переносов (составленные из суммирующих элементов $FA_j^{(i)}$) складывают частичные произведения. Младшие разряды появляются на выходах в первом столбце рисунка, старшие получаются на выходе сумматора в низу рисунка.

Рисунок 29.14 29.14 Матричный умножитель для $n = 4$. Элемент AND, обозначенный $G_j^{(i)}$, вычисляет j-й бит частичного произведения $m^{(i)}$. Каждый горизонтальный ряд суммипующих элементов $FA_j^{(i)}$) представляет собой сумматор с запоминанием переносов. Младшие n битов произведения — это бит $m_0^{(0)}$ и u-биты, получаемые в правом столбце в ходе сложения с запоминанием переносов. Старшие n битов получаются на выходе сумматора, складывающего u-биты v-биты, выходящие из суммирующие элементов нижней строки. Показаны значения для множителей рис. 29.13.

Последовательно выполняемые сложения с запоминанием переносов показаны на рис. 29.15. Вначале из чисел 0, $m^{(0)}$ и $m^{(1)}$ получаются числа $u^{(1)}$ и $v^{(1)}$ (не более $n+1$ битов в каждом; для $v^{(1)}$ это так, поскольку $n+1$-ые разряды двух слагаемых из трёх равны 0), при этом $m^{(0)} + m^{(1)} = u^{(1)} + v^{(1)}$ Затем из чисел $u^{(1)}$, $v^{(1)}$ и $m^{(2)}$ получаются числа $u^{(2)}$ и $v^{(2)}$, имеющие по $n+2$ битов каждое (снова в двух слагаемых из трёх старший бит равен 0). Мы продолжаем процесс, складывая $u^{(i-1)}$, $v^{(i-1)}$ и $m^{(i)}$ для всех $i = 2, 3, \ldots, n-1$. В конце концов мы получаем 2 числа $u^{(n-1)}$ и

v^{n-1} по $(2n-1)$ битов в каждом, при этом

$$u^{(n-1)} + u^{(n-1)} = \sum_{i=0}^{n-1} m^{(i)} = \tag{29.5}$$

$$= p. \tag{29.6}$$

$$\tag{29.7}$$

Не все разряды чисел $v^{(i)}$ используются: поскольку $m_j^{(i)} = 0$ при $j = 0, \ldots, i-1, i+n, \ldots, 2n-1$ и $v_j^{(i)} = 0$ при $j = 0, \ldots, i, i+n, i+n+1, \ldots, 2n-1$ (см. упражнение 29.3-1), на i-м шаге необходимо работать только с $n-1$ разрядами $(i, i+1, \ldots, i+n-2)$. Вернёмся к матричному умножителю (рис. 29.14). Элементы AND (обозначенные $G_j^{(i)}$) вычисляют частичные произведения. Выходом элемента $G_j^{(i)}$ будет бит $m_j^{(i)}$, то есть j-ый бит i-го частичного произведения, так что i-ая строка AND-матрицы даёт n значащих битов частичного произведения $m^{(i)}$ (с индексами $n+i-1, n+i-2, \ldots, i$).

Рисунок 29.15 29.15 Суммирование частичных произведений: повторное сложение с запоминанием переносов. Разобран пример рис. 29.13 ($a = \langle 1110 \rangle, b = \langle 1101 \rangle$). Пустые места перед знаками равенства считаются заполненными нулями. Мы вычисляем $m^{(0)} + m^{(1)} + 0 = u^{(1)} + v^{(1)}$, затем $u^{(1)} + v^{(1)} + m^{(2)} = u^{(2)} + v^{(2)}$, $u^{(2)} + v^{(2)} + m^{(3)} = u^{(3)} + v^{(3)}$, и, наконец, $p = m^{(0)} + m^{(1)} + m^{(2)} + m^{(3)} = u^{(3)} + v^{(3)}$. При этом $p_0 = m_0^{(0)}$ и $p_i = u_i^{(i)}$ для $i = 1, 2, \ldots, n-1$.

Каждый ряд сумматоров $FA_1^{(i)}, \ldots, FA_{n-1}^{(i)}$ совершает i-й шаг сложения с запоминанием переносов, вычисляя $u^{(i)}$ и $v^{(i)}$; элемент $FA_j^{(i)}$ получает на вход биты $m_j^{(i)}$, $u_j^{(i-1)}$ и $v_j^{(i-1}$ и выдаёт два бита $u_j^{(i)}$ и $v_{j+1}^{(i)}$. Заметьте, что в левой колонке $u_{i+n-1}^{(i)} = m_{i+n-1}^{(i)}$, поэтому этот бит не нужно специально вычислять. На входах сумматоров в верхнем ряду фиксировано значение 0, так как одно из слагаемых равно нулю.

Теперь о выходных битах матричного умножителя. Как мы уже отмечали, $v_j^{(n-1)} = 0$ для $j = 0, 1, \ldots, n-1$. Поэтому $p_j = u_j^{(n-1)}$ для $j = 0, 1, \ldots, n-1$. Более того, раз $m_0^{(1)} = 0$, то $u_0^{(1)} = m_0^{(0)}$; поскольку i младших битов у чисел $m^{(i)}$ и $v^{(i-1)}$ равны 0, мы имеем $u_j^{(i)} = u(i-1)_j$ для $i = 2, 3, \ldots, n-1$ и $j = 0, 1, \ldots, i-1$. Следовательно, $p_0 = m_0^{(0)}$ и далее $p_i = u_i^i$ при $i = 1, 2, \ldots, n-1$. Так определяются n младших разрядов произведения. Старшие разряды произведения вычисляются с помощью одной из схем предыдущего раздела:

$$\langle p_{2n-1}, p_{2n-2}, \ldots, p_n \rangle =$$

$$= \langle u_{2n-2}^{(n-1)}, u_{2n-3}^{(n-1)}, \ldots, u_n^{(n-1)} \rangle + \langle v_{2n-2}^{(n-1)}, v_{2n-3}^{(n-1)}, \ldots, v_n^{(n-1)} \rangle.$$

29.3.2 Характеристики схемы

Сложение с запоминанием переносов включает $n-1$ шаг и требует времени $\Theta(n)$. После этого готовы младшие биты произведения и слагаемые для последнего сложения, которое можно выполнять либо за время $\Theta(n)$, либо за время $\Theta(\lg n)$ — последнее не даёт большой экономии, так как суммарное время работы всё равно составляет $\Theta(n)$.

Схема содержит n^2 элементов AND, примерно столько же суммирующих элементов и сумматор размера $\Theta(n)$. Поэтому её размер составляет $\Theta(n^2)$.

29.3.3 Умножение с помощью дерева Уоллеса

Дерево Уоллеса *(Wallace tree) сводит задачу сложения n чисел размера n к сложению двух чисел размера $\Theta(n)$. Это делается так: используя $\lfloor n/3 \rfloor$ сумматоров с запоминанием переносов, мы сводим сложение n чисел к сложению $\lceil 2n/3 \rceil$ чисел. Эта процедура повторяется до тех пор, пока не останется всего два слагаемых. На каждом этапе параллельно работающие сумматоры требуют времени $O(1)$, этапов $\Theta(\lg n)$, последнее сложение (с предвычислением переносов) тоже требует времени $\Theta(\lg n)$, так что общее время будет $\Theta(\lg n)$ при размере $\Theta(n^2)$.*

Дерево Уоллеса для $n=8$ показано на рис. 29.16. На самом деле это не дерево, а ориентированный граф без циклов, но мы сохраняем традиционное название. Мы складываем 8 частичных произведений $m^{(0)}, \ldots, m^{(7)}$, числа на стрелках указывают количество разрядов в складываемых числах ($m^{(i)}$ содержит $n+i$ битов, о числе битов на выходе сумматора с запоминанием переносов см. упражнение 29.3-3). Последнее сложение (слагаемые имеют длины $2n-1$ и $2n$, сумма имеет длину $2n$) выполняется с помощью сумматора с предвычислением переносов.

Рисунок 29.16 29.16 Дерево Wallace'а для сложения частичных произведений $m^{(0)}, \ldots, m^{(7)}$. Каждая стрелка обозначает число указанной разрядности.

29.3.4 Характеристики схемы

Обозначим глубину дерева Уоллеса для n слагаемых через $D(n)$. На каждом уровне из каждых трёх чисел делается по два, да ещё два может остаться (как случилось на первом шаге на *рис. 29.16), так что*

$$D(n) = \begin{cases} 0 & \text{если } n \leqslant 2, \\ 1 & \text{если } n = 3, \\ D(\lceil 2n/3 \rceil) + 1 & \text{если } n \geqslant 4 \end{cases}$$

Отсюда в силу теоремы 4.1 (случай 2) имеем $D(n) = \Theta(\lg n)$. *Частичные произведения могут быть вычислены за время* $\Theta(1)$, *каждый из шагов сложения с запоминанием переносов также требует времени* $\Theta(1)$, *а сложение двух слагаемых размера* $\Theta(n)$ *(последний шаг) требует времени* $\Theta(\lg n)$. *Поэтому общее время работы составляет* $\Theta(\lg n)$.

Покажем теперь, что размер схемы равен $\Theta(n^2)$. *Ясно, что он составляет не менее* $\Omega(n^2)$, *поскольку уже на верхнем ярусе имеется* $\lceil 2n/3 \rceil$ *блоков, каждый из которых содержит не менее* n *сумматоров. Оценим теперь размер схемы сверху. Поскольку результат сложения имеет* $2n$ *разрядов, число сумматоров в каждом из блоков не превосходит* $2n$. *Остаётся показать, что число блоков (обозначим его* $C(n)$*) не превосходит* $O(n)$. *В самом деле,*

$$C(n) = \begin{cases} 1 & \text{если } n = 3, \\ C(\lceil 2n/3 \rceil) + \lfloor n/3 \rfloor & \text{если } n \geqslant 4 \end{cases}$$

откуда в силу теоремы 4.1 (случай 3) следует, что $C(n) = \Theta(n)$. *Таким образом, общее число элементов во всех сумматорах дерева Уоллеса не превосходит* $\Theta(n^2)$. *Так же оценка справедлива для элементов, вычисляющых частичные произведения, а сумматор с предвычислением переносов имеет размер* $\Theta(n)$. *Значит, размер всего умножителя равен* $\Theta(n^2)$.

Хотя умножитель с помощью дерева Уоллеса имеет (асимптотически) тот же размер, что и матричный, и при этом работает (асимптотически) быстрее, он не столь удобен на практике, так как имеет нерегулярную структуру (и элементы трудно плотно разместить на плате). Поэтому используется промежуточный вариант: частичные произведения разбиваются на две половины, каждая складывается с помощью матричного умножителя, из 4 оставшихся чисел делает два с помощью двукрвтного использования сумматора с запоминанием переносов, наконец, два числа складываются с помощью сумматора с предвычислением переносов. Это почти вдвое уменьшает задержку по сравнению с одним матричным умножителем.

29.3.5 Упражнения

29.3-1 Докажите, что в матричном умножителе $v_j^{(i)} = 0$ *при* $j = 0, 1, \ldots, i, i+n, i+n+1, \ldots, 2n-1$.

29.3-2 Измените схему матричного умножителя на рис. 29.14 так, чтобы в верхнем ряду из суммирующих элементов $FA_i^{(1)}$ остался только один.

29.3-3 Пусть при сложении с запоминанием переносов из чисел x, y и z получаются числа s и c. Пусть n_x, n_y, n_x, n_c, n_s — разрядности соответствующих чисел и $n_x \leqslant n_y \leqslant n_z$. Докажите, что $n_s = n_z$ и

$$n_c = \begin{cases} n_z \text{ если } n_y < n_z, \\ n_z + 1 \text{ если } n_y = n_z \end{cases}$$

29.3-4 Показать, что дополнительное ограничение $O(1)$ на выходную степень всех проводов не мешает построить схему умножителя глубины $O(\lg n)$ и размера $O(n^2)$.

29.3-5 Постройте эффективную схему для деления двоичного числа на 3. (Указание. $0.010101\ldots = 0.01 \cdot 1.01 \cdot 1.0001 \cdot \ldots$).

29.3-6 Схема циклического сдвига *(cyclic shifter, barrel shifter) имеет два входа $x = \langle x_{n-1}, x_{n-2}, \ldots, x_0\rangle$ и $s = \langle s_{m-1}, s_{m-2}, \ldots, s_0\rangle$, где $m = \lceil(\lg n)\rceil$ и выход $y = \langle y_{n-1}, y_{n-2}, \ldots, y_0\rangle$, получаемый из x циклическим сдвигом на s позиций вправо: $y_i = x_{(i+s) \bmod n}$ при $(i = 0, 1, \ldots, n-1)$. Как описать эту функцию в терминах умножения остатков?*

29.4 Тактированные схемы

Поскольку схема из функциональных элементов не содержит циклов, каждый элемент в ней используется только один раз. Элементы задержки позволяют использовать один и тот же функциональный элемент многократно (в разные моменты времени), что позволяет уменьшить размер схемы.

В этом разделе рассматриваются тактированные схемы *(clocked circuits) для сложения и умножения. Устройство побитового сложения имеет размер $\Theta(1)$ и складывает два n-разрядных числа за время $\Theta(n)$. Линейный умножитель умножает два n-значных числа за время $\Theta(n)$ и имеет размер $\Theta(n)$.*

29.4.1 Устройство побитового сложения

На рис. 29.17 показана схема для сложения двух n-разрядных чисел $a = \langle a_{n-1}, a_{n-2} \ldots, a_0\rangle$ и $b = \langle b_{n-1}, b_{n-2} \ldots, b_0\rangle$, состоящая только из одного сумматора. Биты поступают на входы последовательно: сначала a_0 и b_0, затем a_1 и b_1, и т. д. При этом выходной бит переноса надо подать на вход сумматора, но не сразу, а в момент поступления следующего разряда.

Такая задержка осуществляется в тактированных схемах

Рисунок 29.17 Устройство побитового сложения. Между i-м и $(i+1)$-м импульсами на входы сумматора *FA* поступают значения a_i и b_i извне и значение c_i из регистра. Сумматор вычисляет значения s_i и c_{i+1}; последнее запоминается в регистре для следующего шага. В начальный момент регистр содержит значение $c_0 = 0$. На рисунках (a)-(e) показаны пять этапов сложения чисел $a = \langle 1011 \rangle$ и $b = \langle 1001 \rangle$

(clocked circuits). Такая схема содержит один или несколько регистров (элементов задержки), а также схему из функциональных элементов, входами которой являются, помимо входов тактированной схемы, выходы регистров. Её выходы служат выходами тактированной схемы, а также входами регистров. Как и раньше, схема из функциональных элементов не должна иметь циклов (обратная связь осуществляется только через элементы задержки).

Элемент задержки, *или* регистр *(register) получает сигналы* тактового генератора *(clock), который через равные промежутки времени выдаёт* тактовые импульсы *(ticks). Мы считаем, что наши схемы имеют общий для всех регистров тактовый генератор (globally clocked circuits).*

При каждом импульсе регистр запоминает текущее значение на входе. Вслед за этим это значение появляется на выходе, и тем самым поступает на входы схемы из фцнкциональных элементов. Выходные значения этой схемы поступают на входы регистров, но не воспринимаются ими до следующего тактового импульса.

Схема, изображённая на рис. 29.17, называется устройством побитового сложения *(bit-serial adder). Чтобы схема работала правильно, период между импульсами должен быть не меньше задержки сумматора (иначе к следующему тику не будет готов выходной бит переноса). Значения на входы подаются с интервалом в один период. В момент первого тактового импульса на входы сумматора подаются значения* a_0*,* b_0 *и* 0 *(рис. 29.17 (a); мы предполагаем, в начальном состоянии на выходе регистра имеется значение* 0*). На выходе через некоторое время появляются бит суммы* s_0 *(который выдаётся наружу) и бит переноса* c_1*. Бит переноса поступает (по проводу) на вход регистра, но лишь после второго импульса появляется на выходе регистра. Вместе со значениями* a_1 *и* b_1 *он даёт на выходе* s_1 *и* c_2 *(рис. 29.17(b)) и т. д. Так продолжается, пока не будут обработаны все разряды слагаемых, после чего на входы подаются нули (*$a_n = 0$*,* $b_n = 0$*), и на выход подаётся старший разряд суммы (совпадающий с последним битом переноса;* $s_n = c_n$*) см. рис. 29.17(e).*

29.4.2 Характеристики схемы

Как мы говорили, тактовая частота зависит от задержки в схеме из функциональных элементов (в данном случае — задержки в элементе суммирования FA), поскольку все вычисления должны закончиться до начала следующего тактового импульса. В данном случае сумматор имеет глубину $\Theta(1)$*, а для получения всех выходов необходимо* $n+1$ *периодов, так что общее время работы составляет* $(n+1)\Theta(1) = \Theta(n)$*. Размер схемы равен* $\Theta(1)$*.*

29.4.3 Каскадное сложение и побитовое сложение

Каскадный сумматор можно рассматривать как развёрнутую схему устройства побитового сложения: теперь за каждый период между импульсами отвечает свой сумматор.

Такое развёртывание, заменяющее время пространством, можно сделать для любой тактированной схемы, соединяя несколько экзмепляров схемы (столько, сколько тактов требует её работа).

Конечно, при этом увеличивается число элементов — зато не нужны синхронизирующие импульсы от тактового генератора. Такая схема на практике работает несколько быстрее, поскольку в тактированной схеме на каждом такте нужно оставлять некоторый запас времени (чтобы вычисления успели произойти), а после развёртывания выходы каскада сразу же попадают на входы следующего, ничего не ожидая.

29.4.4 Одномерный умножитель

Матричный умножитель, рассмотренный в разделе 29.3, имеет размер $\Theta(n^2)$*. В этом разделе мы рассмотрим два тактированных умножителя, использующих одномерный массив элементов (размера* $\Theta(n)$*) вместо двумерного. Более быстный из них имеет время работы* $\Theta(n)$ *(как и матричный умножитель)*

Оба этих умножителя используют алгоритм, который в Америке называют русским народным алгоритмом умножения (Russian peasant's algorithm). Он показан на рис. 29.18 (a). Сомножители a *и* b *записываются рядом, и под каждым из них пишется колонка чисел. В левой колонке число на каждом шаге делится на 2 (остаток отбрасывается); в правой — умножается на 2. Так продолжается до тех пор, пока в левой колонке не будет 1. После этого складываются числа из правой колонки, стоящие напротив нечётных чисел в левой.*

На первый взгляд этот алгоритм кажется удивительным, но он становится понятным, если записать все числа в двоичной си-

стеме: при этом становится ясным, что он представляет собой вариант умножения в столбик. Строки, в которых слева появляется нечётное число, соответствуют единичным разрядам в a, их вклад в сумму есть умноженное на соответствующую степень двойки число b (см. рис. 29.18 (b)).

Рисунок 29.18 29.18 Умножение 19 на 29 с помощью русского народного алгоритма. В колонке a каждое следующее число получается из предыдущего делением на 2 (остаток отбрасывается), а в колонке b -умножением на 2. Складываются числа из правой колонки, напротив которых стоят нечётные числа в левой (выделены серым цветом). (a) Сложение в десятичной системе. (b) То же в двоичной системе.

29.4.5 Простая реализация

Простой вариант реализации русского народного алгоритма в виде тактированной схемы умножения n-битовых чисел показан на рис. 29.19 (a). В нём используется $2n$ групп по 3 регистра. Верхние регистры хранят биты сомножителя a, средние отвечают за сомножитель b, а в нижных постепенно формируется произведение p. Содержимое i-го a-регистра перед j-м шагом обозначается $a_i^{(j)}$; аналогично для b и p. Вместе все биты в a-регистрах образуют двоичную запись числа, которогое обозначется $a^{(j)}$ (по состоянию перед j-м шагом); обозначения $b^{(j)}$, $p^{(j)}$ имеют аналогичный смысл.

Шаги работы нумеруются числами от 0 lj $n-1$. Вначале $a^{(0)} = a$, $b^{(0)} = b$ и $p^{(0)} = 0$. Поддерживается инвариант

$$a^{(j)} \cdot b^{(j)} + p^{(j)} = a \cdot b \qquad (29.6)$$

(см. упражнение 29.4-2). На j-м шаге производятся следующие действия:

1. Если $a_0^{(j)} = 1$ (т. е. $a^{(j)}$ нечётно), то $p^{(j+1)} = p^{(j)} + b^{(j)}$, иначе $p^{(j+1)} = p^{(j)}$.

2. Число a сдвигается вправо на одну позицию (делится на 2 с остатком):

$$a_i^{(j+1)} = \begin{cases} a_{i+1}^{(j)} \text{ если } 0 \leqslant i \leqslant 2n-2 \\ 0 \text{ если } i = 2n-1 \end{cases}$$

3. Число b сдвигается влево на одну позицию (умножается на 2):

$$b_i^{(j+1)} = \begin{cases} a_{i-1}^{(j)} \text{ если } 1 \leqslant i \leqslant 2n-1 \\ 0 \text{ если } i = 0 \end{cases}$$

После n шагов получаем $a^{(n)} = 0$, поэтому инвариант гарантирует, что $p^{(n)} = a \cdot b$.

Наш алгоритм требует n шагов. Если на каждом шаге использовать для сложения каскадный сумматор, то каждый шаг занимает время $\Theta(n)$, поэтому общее время работы составляет $\Theta(n^2)$. Схема состоит из $3n$ регистров, каскадного сумматора размера $\Theta(n)$ и $O(n)$ элементов, формирующих частичные произведения (операция AND над $a_0^{(j)}$ и b-битами). Общий размер схемы равен $\Theta(n)$.

Рисунок 29.19 29.19 Два одномерных умножителя. Показано умножение 5-битовых чисел $a = 19 = \langle 10011 \rangle$ на $b = 29 = \langle 11101 \rangle$ и содержимое регистров перед каждым шагом; значащие биты выделены серым цветом. (a) Простой умножитель (время работы $\Theta(n^2)$). На каждом шаге используется каскадный сумматор, поэтому интервал между тактами должен быть не меньше $\Theta(n)$. (b) Быстрый умножитель, использующий сложение с запоминанием переносов. Каждый шаг требует времени $\Theta(1)$. Всего требуется $2n - 1 = 9$ шагов, показаны первые 5. (После этого остаются сложить u и v, для чего достаточно продолжить тот же процесс сложения с запоминанием переносов.)

29.4.6 Быстрая реализация

Уменьшения времени работы можно добиться, используя вместо каскадного сложения сложение с запоминанием переносов (см. рис. 29.19 (b)). Теперь вместо трёх регистров на каждый разряд будут четыре; в двух из них по-прежнему хранятся цифры чисел a и b, а вместо числа p будт храниться два числа u и v, причём поддерживается инвариант

$$a^{(j)} \cdot b^{(j)} + u^{(j)} + v^{(j)} = a \cdot beqno(29.7)$$

так что роль p играет сумма $u + v$ (см. упражнение 29.4-2). Вначале $u^{(0)} = v^{(0)} = 0$. (Это соответствует использованию алгоритма сложения с запоминанием переносов для хранения промежуточных результатов: на каждом шаге к сумме $u + v$ надо добавить очередное частичное произведение, т. е. надо сложить три числа — и получить ответ в виде суммы двух.)

Шаг алгоритма включает сдвиги a и b (как и в медленном алгоритме), а также изменение u и v. Если $a_0^{(j)} = 1$, то

$$u_i^{(j+1)} = parity(b_i^{(j)}, u_i^{(j)}, v_i^{(j)})$$

при $i = 0, 1, 2, \ldots, 2n - 1$ и

$$v_i^{(j+1)} = \begin{cases} majority(b_{i-1}^{(j)}, u_{i-1}^{(j)}, v_{i-1}^{(j)}) \text{ если } 1 \leqslant i \leqslant 2n-1 \\ 0 \text{ если } i = 0 \end{cases}$$

Если же $a_0^{(j)} = 0$, *то* $b^{(j)}$ *заменяется на* 0:

$$u_i^{(j+1)} = parity(0, u_i^{(j)}, v_i^{(j)})$$

при $i = 0, 1, 2, \ldots, 2n-1$ *и*

$$v_i^{(j+1)} = \begin{cases} majority(0, u_{i-1}^{(j)}, v_{i-1}^{(j)}) \text{ если } 1 \leqslant i \leqslant 2n-1 \\ 0 \text{ если } i = 0 \end{cases}$$

Всего производится $2n-1$ *шаг, причём при* $j \geqslant n$ *имеем* $a^{(j)} = 0$ *и* $u^{(j)} + v^{(j)} = a \cdot b$. *С этого момента сумма* $u + v$ *не меняется, а лишь перераспределяется между* u *и* v. *Наконец,* v *становится равным* 0 *и сумма оказывается в* u: *поскольку* $v^{(2n-1)} = 0$ *(см. упражнение 29-4.3),* $u^{(2n-1)} = a \cdot b$. *(Таким образом, начиная с* n*-го шага по существу происходит сложение двух чисел.)*

Так как каждый шаг занимает время $\Theta(1)$, *общее время работы составляет* $(2n-1)\Theta(1) = \Theta(n)$. *Размер схемы по-прежнему равен* $\Theta(n)$.

Упражнения

29.4-1 Используя русский народный алгоритм, перемножьте $a = \langle 101101 \rangle$ *и* $b = \langle 011110 \rangle$. *Как это выглядит в двоичной и десятичной системах?*

29.4-2 Докажите инварианты (29.6) и (29.7).

29.4-3 Докажите, что при быстрой реализации одномерного умножителя $v^{(2n-1)} = 0$.

29.4-4 Объясните, почему матричный умножитель можно рассматривать как развёрнутый (в пространстве вместо времени) вариант быстрого одномерного умножителя.

29.4-5 Пусть на вход поступают значения $x_1, x_2, \ldots$ *(по одному за такт), а в нашем распоряжении имеется функциональные элементы, вычисляющие максимум своих двух входов, и регистры. Для фиксированного* n *постройте схему размера* $O(n)$, *которая за время* $O(1)$ *вычисляет*

$$y_t = \max_{t-n+1 \leqslant i \leqslant t} x_i$$

(максимум из n *наиболее свежих элементов)*

29.4-6 (Продолжение) То же самое, если есть только* $O(\lg n)$ *элементов "максимум".*

29.5 Задачи

29-1. Схемы для деления

Деление можно свести к сложению и умножению с помощью метода Ньютона *(Newton iteration). Ясно, что достаточно построить схему для вычисления обратного элемента (так как умножитель у нас уже есть).*

Пусть фиксировано число x, и мы хотим вычислить $1/x$. Рассмотрим последовательность $y_0, y_1, y_2, \ldots$, для которой

$$y_{i+1} = 2y_i - xy_i^2$$

Пусть x – n-значная двоичная дробь, $1/2 \leqslant x \leqslant 1$. Мы хотим вычислить $1/x$ с точностью до n знаков (вообще-то $1/x$ скорее всего будет бесконечной десятчиной дробью).

a. *Пусть $|y_i - 1/x| \leqslant \varepsilon$ для некоторого $\varepsilon > 0$. Докажите, что $|y_{i+1} - 1/x| \leqslant \varepsilon^2$.*

b. *Укажите начальное значение y_0, для которого $|y_k - 1/x| \leqslant 2^{-2^k}$ при всех $k \geqslant 0$. Какое k нужно взять, чтобы получить ответ с точностью до n знаков?*

c. *Постройте схему, вычисляющую по n-разрядному числу x число $1/x$ с точностью до n знаков за время $O(\lg^2 n)$. Попытайтесь добиться размера $\Theta(n^2 \lg n)$.*

[Витя: кажется, beat означает 'лучшую чем' - но надо проверить, проявив little cleverness]

29-2. Пропозициональные формулы для симметричных булевых функций.

Функция n аргументов $f(x_1, x_2, \ldots, x_n)$ называется симметричной *(symmetric), если для любой перестановки π чисел $1, 2, \ldots, n$*

$$f(x_1, x_2, \ldots, x_n) = f(x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)})$$

Оказывается, любая симметричная булева функция n аргументов (аргументы и значение принадлежит $\{0, 1\}$) может быть выражена пропозициональной формулой, т. е. формулой, содержащей только $x_1, \ldots, x_n$, скобки и знаки операций $\neg$, $\wedge$ и $\vee$, причём длина этой формулы полиномиально зависит от n. Для начала мы построим для симметричной функции схему глубины $O(\lg n)$, а затем и искомую формулу. Все схемы предполагаются составленными из элементов AND, OR и NOT.

a. *Постройте схему глубины $O(\lg n)$ для для функции* большинства *(majority function)*

$$majority_n(x_1, \ldots, x_n) = \begin{cases} 1 \text{ если } x_1 + x_2 + \ldots + x_n > n/2 \\ 0 \text{ иначе} \end{cases}$$

(Указание: постройте дерево сумматоров.)

b. *Пусть существует схема глубины d, вычисляющая данную булеву функцию f. Покажите, что тогда для f существует формула длины $O(2^d)$ (в частности, для функции большинства существует формула полиномиальной длины).*

c. *Покажите, что симметричная логическая функция может быть записана как функция суммы своих аргументов.*

d. *Докажите, что для каждой симметричной булевой функции n аргументов существует схема глубины $O(\lg n)$.*

e. *Докажите, что для любой симметричной булевой функции n аргументов существует формула, длина которой ограничена полиномом от n (единым для всех функций)*

29.6 Комментарии

Большинство книг по арифметическим схемам обращают больше внимание на практивескую реализацию схем, чем на лежащие в их основе алгоритмы. Книга Сэведжа [173] — одна из немногих книг, посвящённая алгоритмическим вопросам. Легко читаются книги Каванага [39] и Ванга [108] более инженерного содержания. Среди других хороших книг можно назвать книги Хилла и Петерсона [96], а также Кохави [126] (с уклоном в теорию формальных языков).

История арифметических алгоритмов изложена в книге Айкена и Хоппера [7]. Сложение столбикоми появилось не позже абака (которому более 5000 лет). Первое механическое устройство, его реализующее, было создано Б. Паскалем в 1642 году. Вычислительное устройство, позволяющее умножать повторными сложениями, было разработано независимо С. Морландом (S. Morland, 1666) и Г. В. Лейбницем (1671). "Русский народный алгоритм", как пишет Кнут [22], был известен египетским математикам ещё в 1800 г. до н. э.

Идея трёх возможных типов переноса использовалась в релейной вычислительной машине, построенной в Гарварде в 40-е годы нашего века [180]. Алгоритм с предвычислением переносов одними из первых описали Вайнбергер и Смит [199], но их алгоритм требовал элементов с бо́льшим числом входов. Алгоритм сложения двух n-разрядных чисел за время $O(\lg n)$ с элементами, имеющими фиксированное число входов, указал Офман [152]

Идею использовать сложение с запоминанием переносов для умножения высказали Эстрин, Гилхрист и Померен [64]. Атрубин [13] построил бесконечный линейно-матричный умножитель для чисел любой длины (n-й бит произведения выдаётся сразу после поступления n-ых битов сомножителей). Дерево Уоллеса описал Уоллес [197] (независимо его идея была открыта Офманом [152]).

Алгоритмы деления восходят к И. Ньютону (придумавшему свой метод итераций около 1665 года). В задаче 29.1 строится схема для деления, имеющая глубину глубины $O(\lg^2 n)$. В действительности можно построить схему глубины всего $O(\lg n)$ (Бим, *Кук и Хувер [19]).*

30

Алгоритмы параллельных вычислений

С появлением многопроцессорных компьютеров возникла необходимость в разработке параллельных алгоритмов *(parallel algorithms), которые используют возможность одновременного выполнения нескольких действий. Усилиями многих исследователей такие алгоритмы разработаны для множества задач, в том числе и для задач, рассмотренных в предыдущих главах. В этой главе мы рассмотрим несколько простых параллельных алгоритмов в качестве иллюстрации основных идей.*

Сортирующие сети и схемы из функциональных элементов (главы 28, 29) представляют собой модель параллельных вычислений, но недостаточно общую. В этой главе в качестве модели используются "параллельные машины с произвольным доступом". Они удобны для описания алгоритмов работы со структурами данных (массивами, деревьями, списками и т. д.). Эта модель достаточно близка к практике в том смысле, что если один алгоритм оказывается эффективнее другого для этой модели, то, как правило, он будет эффективнее и при практической реализации.

30.0.1 Параллельная машина с произвольным доступом (PRAM)

Устройство параллельной машины с произвольным доступом *(parallel random-access machine, PRAM) показано на рис. 30.1. Процессоры* $P_0, \dots, P_{p-1}$ *используют общую память (shared memory). Все* p *процессоров могут одновременно записывать информацию в память или читать из памяти, а также параллельно выполнять арифметические и логические операции.*

В качестве меры для оценки времени работы мы выбираем число циклов параллельного доступа к общей памяти, считая, что

Рисунок 30.1 30.1. Устройство PRAM-машины. Процессоры $P_0, P_1, \dots, P_{p-1}$ используют общую память. Каждый процессор может получить доступ к любой ячейке памяти за единичное время.

каждый такой цикл занимает фиксированное время. На практике это не совсем так, поскольку время доступа к памяти зависит от количества используемых процессоров. Тем не менее, в реальных многопроцессорных компьютерах доступ к памяти (обычно с помощью специальной сети коммутации) действительно занимает большую часть времени, так что число обращений к памяти является разумной оценкой времени работы.

При оценке алгоритмов для PRAM необходимо учитывать не только время работы, но и количество используемых процессоров. В этом состоит важное отличие от однопроцессорной модели, где мы интересуемся в основном временем работы. Как правило, уменьшение числа процессоров приводит к соответствующему увеличению времени работы. Этот вопрос обсуждается в разделе 30.3.

30.0.2 Параллельный и исключительный доступ к памяти

Существует несколько подходов к использованию общей памяти. В алгоритмах с одновременным чтением *(concurrent-read algorithms) несколько процессоров могут одновременно считывать информацию из одной ячейки. В* алгоритмах с исключающим чтением *(exclusive-read algorithms) одновременное чтение (из одной ячейки памяти) запрещено. Подобным же образом алгоритмы делятся на* алгоритмы с одновременной записью *(concurrent-write algorithms) и* алгоритмы с исключающей записью *(exclusive-write algorithms). Таким образом, возможно четыре вида общей памяти, для которых традиционно употребляются следующие названия:*

- *EREW (exclusive-read exclusive-write): исключающее чтение и исключающая запись,*
- *CREW: одновременное чтение и исключающая запись,*
- *ERCW: исключающее чтение и одновременная запись,*
- *CRCW: одновременное чтение и одновременная запись.*

Далее мы будем говорить, например, о EREW-машине или CREW-алгоритме, имея в виду соответствующий тип общей памяти.

Наиболее употребительны модели EREW и CRCW. Ясно, что все алгоритмы, которые могут выполняться на EREW-машине, могут выполняться и на CRCW-машине, но не наоборот. Аппаратная поддержка модели EREW проще и быстрее (из-за отсутствия конфликтов при записи и чтении). Реализация модели CRCW более сложна, если мы хотим обеспечить хотя бы примерно одинаковое время доступа к памяти для всех вариантов доступа, но с точки зрения программиста эта модель значительно проще и удобнее.

Из двух оставшихся типов (CREW, ERCW) более популярна модель CREW, хотя на практике реализовать одновременную запись не сложнее, чем одновременное чтение. В этой главе считается, что все алгоритмы, использующие одновременное чтение или запись (хотя бы одно из двух) предназначены для CRCW-машины. Мы вернёмся к сравнению различных моделей в в разделе 30.2.

При рассмотрении одновременной записи следует уточнить, что будет находиться в ячейке после записи. В этой главе используется модель одновременной записи общего значения *(common-CRCW model), в которой все процессоры, производящие запись в данную ячейку, обязаны записывать туда одно и то же значение. В литературе встречаются и другие модели:*

- произвольный выбор *(arbitrary): сохраняется одно (произвольное) значение из записываемых;*
- приоритеты *(priority): сохраняется значение, поступившее от процессора с наименьшим номером;*
- комбинация *(combinational): сохраняется некоторая комбинация (например, сумма или максимум) поступивших на запись значений.*

В последнем случае сохраняемое значение обычно является коммутативной и ассоциативной функцией значений, поступающих на запись.

30.0.3 Синхронизация

Для правильной работы параллельных алгоритмов работу процессоров следует синхронизировать. Кроме того, часто требуется проверять условия, зависящие от состояния всех процессоров сразу, причём желательно делать это за постоянное время, не зависящее от числа процессоров. На практике синхронизация и проверки осуществляются с помощью специальной управляющей сети, соединяющей все процессоры. Эта сеть работает примерно с той же скоростью, что и сеть, обеспечивающая доступ к общей памяти.

В дальнейшем мы предполагаем, что с помощью управляющей сети за время $O(1)$ можно проверить какое-либо условие (например, условие окончания цикла) одновременно во всех процессорах (цикл завершается, если все процессоры с этим согласны). Иногда рассматривают EREW-модели без такого механизма, и тогда подобная проверка занимает логарифмическое время, которое должно быть включено в общее время работы (см. упр. 30.1-8). В разделе 30.2 будет показано, что на CRCW-машине условие окончания можно проверить за время $O(1)$, не используя управляющую сеть (с помощью одновременной записи).

30.0.4 План главы

В разделе 30.1 рассматривается метод переходов по указателям и его использование для обработки префиксов списка, а также аналогичные методы для работы с деревьями. Раздел 30.2 посвящён сравнению CRCW- и EREW-машин и преимуществам одновременного доступа к памяти.

В разделе 30.3 доказывается теорема Брента о моделировании схем из функциональных элементов в модели PRAM. Кроме этого, в этом разделе рассматривается понятие эффективности по затратам и даются условия, при которых число процессоров может быть уменьшено без потери эффективности. В разделе 30.4 вновь рассматривается задача обработки префиксов списка и описывается эффективный по затратам вероятностный алгоритм для этой задачи. Наконец, раздел 30.5 посвящён задаче о нарушении симметрии. Приводится детерминированный алгоритм, который позволяет решить эту задачу за время, существенно меньшее логарифмического.

Параллельные алгоритмы, рассмотренные в этой главе, по существу относятся к теории графов. Это всего лишь одна из многих областей, где параллельные алгоритмы применяются — но на этом примере мы иллюстрируем достаточно общие идеи и методы.

30.1 Переходы по указателям

В этом разделе рассматривается метод переходов по указателям — важное средство для построения параллельных алгоритмов обработки списков. С помощью этого метода строится алгоритм, позволяющий за время $O(\lg n)$ найти положение в списке (расстояние до конца списка) для всех элементов списка длиной n. Похожий алгоритм используется для параллельной обработки префиксов списка длиной n за время $O(\lg n)$. Мы укажем также способ, позволяющий сводить многие задачи о деревьях к задачам о списках, решаемым с помощью переходов по указателям. Все алгоритмы в этом разделе не требуют одновременного доступа к памяти и потому могут быть реализованы на EREW-машине.

30.1.1 Номер в списке

Список в PRAM-машине хранится обычным образом. При этом мы будем считать, что за каждый элемент списка отвечает свой процессор, то есть что i-ый процессор отвечает за элемент списка, хранящийся в i-ой ячейке памяти. (Порядковый номер

Рисунок 30.2 30.2. Использование метода переходов по указателям для вычисления расстояний до конца списка за время $O(\lg n)$. (a). Начальное состояние: все значения d равны 1. (b)–(d). Состояния списка после каждого выполнения цикла **while** в алгоритме List-Rank. В конце значение d совпадает с расстоянием до конца списка.

этого элемента в списке может быть любым.) На рис. 30.2(a) показан пример списка из объектов $\langle 3, 4, 6, 0, 1, 5\rangle$. *Поскольку для каждого элемента имеется собственный процессор, все элементы списка можно обрабатывать параллельно.*

Пусть дан односторонне связанный список L из n элементов, и для каждого его элемента требуется найти расстояние до конца списка. Более формально, если $next[i]$ – указатель объекта i на следующий объект, то расстояние до конца списка задаётся индуктивно следующим образом:

$$d[i] = \begin{cases} 0 & \textit{если } next[i] = \text{NIL} \\ d[next[i]] + 1 & \textit{если } next[i] \neq \text{NIL} \end{cases}$$

Эта задача называется задачей о номере в списке *(list-ranking problem).*

Тривиальное решение состоит в вычислении расстояний последовательно, начиная с конца списка. При этом время работы составляет $\Theta(n)$, поскольку k-й с конца объект может быть обработан лишь после $k-1$ следующих за ним объектов. Такое решение по сути является последовательным, а не параллельным — в каждый момент активен только один процессор. Рассмотрим теперь алгоритм с временем работы $\Theta(\lg n)$.

```
List-Rank(L)
1 for (для) каждого процессора
2     do if $next[i] = \text{NIL}$
3         then $d[i] \leftarrow 0$
4         else $d[i] \leftarrow 1$
5 while существует объект $i$, для которого $next[i] \neq \text{NIL}$
6     do for (для) каждого процессора $i$
7         do if $next[i] \neq \text{NIL}$
8             then $d[i] \leftarrow d[i] + d[next[i]]$
9                  $next[i] \leftarrow next[next[i]]$
```

На рис. 30.2 показаны состояния списка перед каждым повторением цикла **while** *в строках 5–9. Состояние списка после заполнения полей $d[i]$ (строки 1–4) показано на рис. 30.2(a). На первом шаге у всех объектов, кроме последнего, указатели не равны* NIL, *поэтому строки 8–9 выполняются для первых пяти процессоров. Получается состояние списка, показанное на рис. 30-2(b). На следующем шаге строки 8–9 выполняются только для первых четырёх процессоров (у двух последних объектов указатели уже равны* NIL*). Результат показан на рис. 30-2(c). При третьем (и* *последнем) выполнении цикла работают только два первых процессора. Конечный результат показан на рис. 30-2(d).*

Идея композиции переходов по указателям *(pointer jumping) реализована в строке 9, где производится операция $next[i] \leftarrow next[next[i]]$. Заметьте, что эти действия нарушают структуру списка, поскольку меняются поля указателей. Если необходимо сохранить список, необходимо сделать копии полей указателей и производить действия с копиями.*

30.1.2 Корректность

Цикл в строках 5–9 имеет следующий инвариант: для любого объекта i либо $next[i]$ указывает на объект, находящийся в списке на расстоянии $d[i]$, либо $next[i] = \text{NIL}$ и $d[i]$ есть расстояние до конца списка.

Заметьте, что для правильной работы алгоритма необходима синхронизация: чтение всех указателей $next[next[i]]$ (в строке 9) должно произойти до записи их новых значений.

Теперь покажем, что программа List-Rank *не использует одновременного обращения к памяти (и поэтому может выполняться на EREW-машине). Ясно, что в строках 2–7, а также при записи в строках 8 и 9 не происходит одновременного обращения к одной ячейке, так как за каждый элемент списка отвечает свой процессор. Кроме того, переход по указателям сохраняет следующий инвариант: для любых двух различных объектов i и j либо $next[i] \neq next[j]$, либо $next[i] = next[j] = \text{NIL}$. Действительно, вначале это условие выполнено; оно сохраняется при исполнении строки 9. Поэтому при чтении в строке 9 не происходит одновременного обращения к одной ячейке.*

В строке 8 не происходит одновременного чтения благодаря синхронизации: мы считаем. что сначала все процессоры читают $d[i]$, и лишь затем $d[next[i]]$. В результате при $i = next[j]$ содержимое $d[i]$ сначала прочитывается i-м процессором, а затем j-м. При такой синхронизации алгоритм может работать на EREW-машине.

В дальнейшем предполагается, что при выполнении программ предусмотрена надлежащая синхронизация, и подобного рода требования всегда выполнены.

30.1.3 Анализ

Пусть список L содержит n элементов. Покажем, что время работы алгоритма составляет $O(\lg n)$. Поскольку инициализация и каждое выполнение цикла **while** *занимают время $O(1)$, достаточно показать, что цикл выполняется $\lg n$ раз. Это следует из* *того, что при каждом выполнении цикла значения $d[i]$ для тех элементов i, для которых $next[i] \neq$ NIL, увеличиваются вдвое.*

Мы предполагаем, что проверка условия в строке 5 занимает время $O(1)$ (при использовании управляющей сети). В упражнении 30.1-8 предлагается реализовать эту проверку без такой сети, сохранив время работы программы $O(\lg n)$.

Кроме времени работы, для параллельных алгоритмов мы будем оценивать также общие затраты *(work), то есть произведение времени работы на количество используемых процессоров. Эта величина оценивает сверху время работы (однопроцессорного) алгоритма, моделирующего работу PRAM путём последовательного моделирования работы всех процессоров.*

Общие затраты на вычисление номеров в списке составляют $\Theta(n \lg n)$. Поскольку однопроцессорный алгоритм работает за время $\Theta(n)$ (за один проход по списку можно вычислить расстояния элементов от начала списка, а также его длину), при выполнении алгоритма List-Rank *процессоры выполняют в сумме в $\lg n$ раз больше действий, чем необходимо однопроцессорному алгоритму.*

Говорят, что алгоритм A не менее эффективен по затратам, чем алгоритм *B (A is work-efficient with respect to B), если общие затраты A превышают общие затраты B не более чем в константу раз. Алгоритм A называется* эффективным по затратам *(work-efficient), если он не менее эффективен, чем любой однопроцессорный алгоритм. (В этом случае он не менее эффективен, чем любой алгоритм, поскольку многопроцессорный алгоритм можно моделировать на одном процессоре, последовательно моделируя работу каждого процессора.) Рассмотренный выше алгоритм отыскания номера в списке не является эффективным по затратам, поскольку существует алгоритм с затратами $\Theta(n)$. Эффективный по затратам вероятностный алгоритм для этой задачи строится в разделе 30.4.*

30.1.4 Параллельная обработка префиксов списка

Метод переходов по указателям применяется не только для определения порядковых номеров элементов списка. Как показано в разделе 29.2.2, обработка префиксов позволяет быстро складывать числа. Здесь будет показано, как использовать метод переходов по указателям для обработки префиксов. Алгоритм обрабатывает префиксы n-элементного списка на EREW-машине за время $O(\lg n)$.

Пусть $\otimes$ — ассоциативная бинарная операция. Задача обработки префиксов *(prefix computation) состоит в следующем: по данной последовательности $\langle x_1, x_2, \ldots, x_n \rangle$ построить последова-*

тельность $\langle y_1, y_2, \ldots, y_n \rangle$, для которой $y_1 = x_1$ и

$$y_k = y_{k-1} \otimes x_k = x_1 \otimes x_2 \otimes \ldots \otimes x_k$$

для $k = 2, 3, \ldots, n$. Другими словами, y_k есть "$\otimes$-произведение" первых k элементов последовательности.

Мы будем считать, что элементы $x_1, \ldots, x_n$, подлежащие обработке, связаны в односторонний список. (Аналогичная задача для массива рассматривается в упражнении 30-1.2). Заметим для начала, что задачу о номере в списке можно рассматривать как частный случай задачи о параллельной обработке префиксов. Действительно, пусть каждый элемент списка равен 1, а операция $\otimes$ — обычное сложение. Тогда $y_k = k$, то есть мы вычислили расстояние до начала списка. Поэтому обработку префиксов можно использовать для решения задачи о номере в списке — нужно лишь сначала обратить список (что можно сделать за время $O(1)$).

Вернёмся к задаче о параллельной обработке префиксов. Введём следующие обозначения:

$$[i, j] = x_i \otimes x_{i+1} \otimes \ldots \otimes x_j$$

при $1 \leqslant i \leqslant j \leqslant n$. В частности, $[k, k] = x_k$ и

$$[i, k] = [i, j] \otimes [j + 1, k]$$

при $1 \leqslant i \leqslant j < k \leqslant n$. Наша цель — вычислить $y_k = [1, k]$ для всех $k = 1, \ldots, n$.

В программе через $x[i]$ обозначается значение объекта i. (Напоминаем, i есть адрес в памяти, а также номер ответственного за эту ячейку процессора, и никак не связан с k — порядковым номером в списке!) Алгоритм вычисляет $y[i] = y_k = [1, k]$, где $x[i] = x_k$ есть k-й с начала элемент списка.

```
List-Prefix(L)
for (для) каждого процессора $i$
    do $y[i] \leftarrow x[i]$
while существует объект $i$, для которого $next[i] \neq$ NIL
    do for (для) каждого процессора $i$
        do if $next[i] \neq$ NIL
            then $y[next[i]] \leftarrow y[i] \otimes y[next[i]]$
                 $next[i] \leftarrow next[next[i]]$
```

Видно, что алгоритм похож на алгоритм определения порядковых номеров элементов. Отличаются лишь инициализация и изменение значений y (в предыдущем алгоритме d). В алгоритме List-Rank процессор i изменяет значение $y[i]$, соответствующее своему объекту, тогда как в алгоритме List-Prefix i-й процессор изменяет "чужое" значение $y[next[i]]$. (Если бы мы обрабатывали

Рисунок 30.3 30.3. Работа алгоритма List-Prefix. (a). Вначале для k-го объекта значение y равно $[k,k]$, а указатель $next[k]$ указывает на $(k+1)$-й объект (указатель последнего – на nil). (b)-(d). Значения y и $next$ после каждого выполнения цикла **while** (строки 3–6). В конце алгоритма (d) для k-ого объекта значение y равно $[1,k]$.

не префиксы, а суффиксы, то этой разницы бы не было.) Как и алгоритм List-Rank, *алгоритм* List-Prefix *поддерживает следующий инвариант: для любых двух различных объектов i и j либо $next[i] \neq next[j]$, либо $next[i] = next[j] =$* nil. *Поэтому не происходит одновременного обращения к памяти, так что алгоритм можно реализовать на EREW-машине.*

На рис. 30.3 показано состояние списка перед каждым выполнением цикла **while**. *Цикл имеет следующий инвариант: после t-го повторения цикла указатели охватывают 2^t звеньев списка (или равны* nil*), а k-й по списку элемент хранит значение $[\max(1, k-2^t+1), k]$ для $k = 1, 2, \ldots, n$. Перед первым выполнением цикла ($t=0$) каждый объект (кроме последнего) указывает на следующий. В строке 6 алгоритма k-й элемент (точнее, отвечающий за него процессор) вычисляет значение $[k, k+1] = [k,k] \otimes [k+1, k+1]$, которое передаётся $(k+1)$-му элементу. Указатели меняются так же, как и в алгоритме* List-Rank. *Состояние списка после первого выполнения цикла показано на рис. 30.3 (b). При втором выполнении цикла k-й элемент (при $k = 2, 3, \ldots, n-2$) вычисляет значение $[k-1, k+2] = [k-1, k] \otimes [k+1, k+2]$, которое передаётся $k+2$-му элементу; результат показан на рис. 30.3 (c). Наконец, в ходе третьего (последнего) выполнения цикла оставшиеся два элемента вычисляют нужные значения (рис. 30-3 (d)).*

Как и предыдущий алгоритм, алгоритм List-Prefix *работает за время $O(\lg n)$, а общие затраты составляют $O(n \lg n)$.*

30.1.5 Метод эйлерова цикла

В этом разделе рассмотрен метод эйлерова цикла и его использование в задаче вычисления глубины вершин двоичного дерева. Если дерево имеет n вершин, то вычисление их глубин займёт время $O(\lg n)$; при этом будет использована техника параллельной обработки префиксов.

Дерево хранится в памяти так, как описано в разделе 11.4: каждая вершина i имеет поля $parent[i]$ (родитель), $left[i]$ (левый ребёнок) и $right[i]$ (правый ребёнок).

Имея один процессор, можно вычислить глубину всех вершин дерева с n вершинами за время $O(n)$. Как сократить это время, используя несколько процессоров? Простейший параллельный алгоритм состоит в движении от корня дерева к листьям с увели-

чением счётчика глубины. При этом вершина глубины k достигается в момент времени k, так что время работы пропорционально высоте дерева. Для полного дерева этот алгоритм требует логарифмического времени, но бывают деревья, у которых высота равна $n-1$. Метод эйлерова цикла, который мы сейчас изложим, позволяет обработать любое дерево с n вершинами за время $O(\lg n)$ (независимо от его высоты).

Эйлеровым циклом *в графе называется путь, проходящий ровно один раз по каждому ребру (вершины могут проходиться многократно). Согласно задаче 23-2, связный ориентированный граф имеет эйлеров цикл в том и только в том случае, когда у каждой вершины входная и выходная степени совпадают. Если превратить связный неориентированный граф в ориентированный, заменив каждое ребро двумя противонаправленными, то такой граф, очевидно, будет иметь эйлеров цикл.*

Пусть T – двоичное дерево. Будем предполагать, что его вершины занумерованы неотрицательными целыми числами. С каждой вершиной мы связываем три процессора A, B и C; узлу с номером i соответствуют процессоры с номерами $3i$, $3i+1$ и $3i+2$. Рассмотрим ориентированный вариант дерева (с удвоенными рёбрами). Эйлеров цикл (или обход дерева) можно превратить в список (см. рис. 30.4(a)):

- *A-процессор вершины указывает на A-процессор левого ребёнка, если он есть, а иначе на B-процессор той же вершины.*
- *B-процессор вершины указывает на A-процессор правого ребёнка, если он есть, а иначе на C-процессор той же вершины.*
- *C-процессор вершины указывает на B-процессор родителя, если это левый ребёнок, и на C-процессор родителя, если это правый ребёнок. C-процессор корневой вершины указывает на* NIL.

Таким образом, список начинается в A-процессоре корневой вершины и заканчивается в её C-процессоре. Этот список может быть построен по данному дереву за время $O(1)$.

Глубину каждой вершины можно подсчитать как сумму изменений глубин по пути к ней (по эйлеровому циклу). Более формально, пусть в каждом A-процессоре находится число 1, в каждом B-процессоре — число 0, а в каждом C-процессоре — число (-1). Произведём параллельное вычисление префиксов, используя в качестве операции обычное сложение. Его результат для дерева рис. 30.4(a) показан на рис. 30.4(b).

После этого C-процессор каждой вершины будет содержать её глубину, а A- и B-процессоры — на единицу большее значение. Это можно проверить, двигаясь вдоль эйлерова цикла. В самом деле, к A-процессору ведёт стрелка вниз по дереву с началом в A- или B-процессоре родителя; к B-процессору ведёт либо стрелка вверх

Рисунок 30.4 30.4. Использование метода эйлерова цикла для вычисления глубины вершин двоичного дерева. (a) Эйлерову циклу (обходу дерева) соответствует список. Каждый процессор хранит число, которое используется при параллельном вычислении префиксов. (b) Результат вычисления префиксов для дерева на рисунке (a). C-процессор каждой вершины (зачернённый) содержит её глубину.

из C-процессора левого потомка, либо стрелка из A-процессора той же вершины; к C-процессору ведёт либо стрелка вверх из C-процессора правого потомка, либо стрелка из B-процессора той же вершины; во всех случаях наше утверждение остаётся верным.

Создание списка занимает время $O(1)$, а параллельная обработка префиксов — время $O(\lg 3n)$, т. е. $O(\lg n)$. Таким образом, общее время работы составляет $O(\lg n)$. Поскольку параллельная обработка префиксов не требует одновременного доступа к памяти, алгоритм может быть реализован на EREW-машине.

Упражнения

30.1-1. Укажите EREW-алгоритм, который для каждого элемента n-элементного списка определяет, является ли он средним ($\lfloor n/2 \rfloor$-м), за время $O(\lg n)$.

30.1-2. Постройте EREW-алгоритм для обработки префиксов массива $x[1\ldots n]$ за время $O(\lg n)$. (Можно обойтись без указателей)

30.1-3. Пусть в списке L имеются объекты двух разных типов (некоторые красные, а остальные синие). Укажите эффективный EREW-алгоритм, формирующий из этого списка два: один из синих объектов, а другой из красных (с сохранением взаимного расположения).

30.1-4. Пусть имеется несколько непересекающихся кольцевых списков, содержащих в совокупности n элементов. Укажите эффективный EREW-алгоритм, который выбирает по одному представителю из каждого списка и сообщает каждому элементу, кто является представителем его списка. (Считается, что процессоры занумерованы и каждому процессору известен его собственный номер.)

30.1-5. Найдите EREW-алгоритм, который за время $O(\lg n)$ определяет для каждой вершины размер поддерева с корнем в этой вершине. (Указание: следует взять разность двух префиксов суммы вдоль эйлерова цикла).

30.1-6. Вершины дерева можно нумеровать в разном порядке. Рассмотрим три способа: корень — левое поддерево — правое поддерево (preorder); левое поддерево — корень — правое поддерево (inorder); левое поддерево — правое поддерево — корень (postorder). Укажите для каждого из способов алгоритм, вычисляющий номера всех вершин.

30.1-7. Распространите метод эйлерова цикла на деревья с произвольной степенью вершин (выбрав подходящее представление дерева). Постройте алгоритм, вычисляющий глубину всех n вершин дерева (в этом представлении) за время $O(\lg n)$.

30.1-8. Измените алгоритм List-Rank так, чтобы проверка условия окончания цикла производилась явно (без использования управляющей сети), сохранив оценку на работы. (Указание: следите за первым элементом списка).

30.2 CRCW- и EREW-алгоритмы

Одновременный доступ к памяти в параллельных компьютерах имеет свои плюсы и минусы. Недостатком является сложность аппаратной поддержки такого доступа. Кроме того, возможности одновременного доступа используются не так часто. С другой стороны, в некоторых случаях без этих возможностей трудно обойтись. На практике часто выбирается один из промежуточных вариантов.

В этом разделе рассматриваются преимущества одновременного доступа. Существуют задачи, для которых наилучший CRCW-алгоритм работает быстрее наилучшего EREW-алгоритма. Такова, например, задача отыскания корней деревьев в графе, являющемся лесом, а также задача поиска максимального элемента в массиве. Эти задачи рассмотрены ниже.

30.2.1 Польза параллельного чтения

Пусть дано несколько двоичных деревьев, заданных следующим образом: каждая вершина i имеет указатель $parent[i]$ на родителя (если вершина является корнем, то $parent[i]$ = NIL). Требуется для каждой вершины i найти корень $root[i]$ дерева, которому она принадлежит. С каждой вершиной мы связываем один процессор.

```
Find-Roots(F)
1 for(для) каждого процессора $i$
2    do if $parent[i]$ = NIL
3          then $root[i] \leftarrow i$
4 while существует узел $i$, для которого $parent[i] \neq$ NIL
5    do for(для) каждого процессора $i$
6          do if $parent[i] \neq$ NIL
7                then $root[i] \leftarrow root[parent[i]]$
8                     $parent[i] \leftarrow parent[parent[i]]$
```

Работа алгоритма показана на рис. 30.5. После инициализации (строки 1–3) корни известны только для корневых вершин (рис. 30.5 (а)). В цикле **while** (строки 4–8) происходят переходы

Рисунок 30.5 30.5. CREW-алгоритм нахождения корней деревьев. Номера вершин расположены рядом с ними. Внутри вершин показаны значения *root*. Стрелки обозначают ссылки *parent*. (a)–(d) Состояние дерева перед каждым выполнением цикла **while** (строки 4–8). Заметим, что максимальная длина пути по стрелкам на каждом шаге уменьшается вдвое.

по указателям и заполняются поля root у вершин следующих (по глубине) уровней. На рис. 30.5 (b)–(d) показаны состояния деревьев после первого, второго и третьего выполнений цикла. Цикл имеет следующий инвариант: после t повторений цикла либо $parent[i] =$ NIL и $root[i]$ содержит правильный указатель на корень, либо $parent[i]$ есть предок на расстоянии 2^t.

Алгоритм FIND-ROOTS *можно реализовать на CREW-машине; время работы составляет $O(\lg d)$, где d — максимальная из глубин деревьев. Действительно, все записи являются исключающими — каждый процессор производит запись только в поля своей вершины. Однако чтение в строках 7 и 8 является одновременным, поскольку одна вершина может быть предком сразу для нескольких других. Например, на рис. 30.5 (b) при втором выполнении цикла значения $root[4]$ и $parent[4]$ читаются процессорами 18, 2 и 7 одновременно.*

Время работы составляет $O(\lg d)$ по тем же причинам, что и для алгоритма LIST-RANK.

Пусть теперь одновременное чтение запрещено. Тогда можно показать, что не существует алгоритма со временем работы меньше $\Omega(\lg n)$ (где n — общее количество вершин). Причина тут в том, что на каждом шаге число вершин, которые знают свой корень, увеличивается не более чем вдвое, поэтому требуется $\Omega(\lg n)$ шагов (для леса из одного дерева).

Если максимальная высота деревьев мала, то алгоритм FIND-ROOTS *работает асимптотически быстрее, чем любой EREW-алгоритм. Например, при высоте дерева $d = O(\lg n)$ (пусть, скажем, имеется всего одно полное дерево), CREW-алгоритм работает за время $O(\lg\lg n)$, а любой EREW-алгоритм требует времени $O(\lg n)$.*

Другой (более простой) пример выигрыша от параллельного чтения см. в упр. 30.2-1.

30.2.2 Польза параллельной записи

Рассмотрим задачу о нахождении максимального элемента в массиве из n действительных чисел. Можно доказать, что для этой задачи наилучший EREW-алгоритм требует времени $\Omega(\lg n)$ и что одновременное чтение не позволяет уменьшить время работы. Однако существует CRCW-алгоритм со временем работы

Рисунок 30.6 30.6. Отыскание максимума в массиве за время $O(1)$ с помощью алгоритма Fast-Max. Для каждой (упорядоченной) пары $\langle i,j\rangle$ элементов массива $A = \langle 5,6,9,2,9\rangle$ в таблице показан результат сравнения $A[i] < A[j]$ (T означает TRUE, F — FALSE). Если в строке есть хотя бы одна буква T, то соответствующий элемент m равен FALSE, иначе — TRUE. Элементы массива, для которых в столбце m стоит значение TRUE, являются максимальными (и записываются в ячейку max).

$O(1)$. Напомним, что мы рассматриваем CRCW-модель, в которой все процессоры, производящие запись в данную ячейку, должны записывать туда одно и то же.

Пусть $A[0\ldots n-1]$ – входной массив. Наш CRCW-алгоритм использует n^2 процессоров. Каждый процессор сравнивает какие-то два элемента $A[i]$ и $A[j]$, где $0 \leqslant i,j \leqslant n-1$, и нумеруется парой индексов (i,j).

```
Fast-Max(A)
1 $n \leftarrow length[A]$
2 for $i \leftarrow 0$ to $n-1$
3     do $m[i] \leftarrow$ TRUE
4 for $i \leftarrow 0$ to $n-1$ и $j \leftarrow 0$ to $n-1$ (параллельно)
5     do if $A[i] < A[j]$
6         then $m[i] \leftarrow$ FALSE
7 for $i \leftarrow 0$ to $n-1$ (параллельно)
8     do if $m[i] =$ TRUE
9         then $max \leftarrow A[j]$
10 return $max$
```

В строке 1 определяется длина массива A (каким-то из процессоров). В строках 2–3 каждым элементом массива m занимается какой-то один процессор (их у нас достаточно — n^2) и помещает туда значение TRUE. (Будущий смысл массива m таков: $m[i]$ истинно, когда $A[i]$ — максимальный элемент в массиве).

Дальнейшая работа алгоритма показана на рис. 30.6. В строках 4–6 происходит сравнение всех возможных пар $A[i]$ и $A[j]$. Если $A[i] < A[j]$, то $A[i]$ не может быть максимальным элементом, поэтому мы записываем в $m[i]$ значение FALSE (строка 6). Несколько различных процессоров могут одновременно производить запись в ячейку $m[i]$, но все они записывают одно и то же значение - FALSE.

Таким образом, после выполнения строк 4–6 $m[i]$ истинно для тех и только тех индексов i, для которых $A[i]$ — максимальный элемент. В строках 7–9 максимальное значение помещается в переменную max, которая является выходным значением (строка 10). К переменной max могут обращаться сразу несколько процессоров, но все они присваивают ей одно и то же значение, как и требуется в модели одновременной записи общего значения.

Поскольку выполнение каждого из циклов занимает время $O(1)$

(все процессоры работают одновременно), общее время работы алгоритма составляет $O(1)$. Алгоритм не является эффективным по затратам: общие затраты составляют $n^2 \cdot O(1) = O(n^2)$, а простейший однопроцессорный алгоритм работает за время $\Theta(n)$. Более эффективный алгоритм рассматривается в упражнении 30.2-6.

В сущности, алгоритм поиска максимума основан на возможности вычислить логическое И от n аргументов за время $O(1)$ при наличии n процессоров в модели с одновременной записью общего значения (и в других, более мощных CRCW-моделях). Фактически для $i = 0, 1, \ldots, n-1$ вычисляется

$$m[i] = \bigwedge_{j=0}^{n-1} (A[i] \geqslant A[j])$$

Возможность быстро вычислять логическое И от многих аргументов используется и в других ситуациях. Например, в ранее рассмотренных EREW-алгоритмах можно обойтись без использования управляющей сети при проверке условия окончания цикла, зависящего от состояния всех процессоров, если это условие есть конъюнкция условий, проверяемых в каждом из процессоров за время $O(1)$.

EREW-машина не обладает такими возможностями: любой EREW-алгоритм нахождения максимального элемента в массиве требует времени $\Omega(\lg n)$. Причина тут в том, что после каждого шага алгоритма число потенциально максимальных элементов уменьшается не более чем вдвое, поскольку для того, чтобы исключить элемент из числа "подозреваемых", нужно, чтобы он оказался меньше какого-то другого. Следовательно, каждое сравнение отбрасывает максимум один элемент, а число сравнений за один шаг не может быть больше половины числа оставшихся под подозрением элементов.

Интересно, что нижняя оценка $\Omega(\lg n)$ сохраняется, даже если разрешить одновременное чтение, то есть любой CREW-алгоритм также требует времени $\Omega(\lg n)$. Кук, Дворк и Райшук [50] показали, что время работы CREW-алгоритма не может быть меньше $\Omega(\lg n)$ даже при наличии неограниченного числа процессоров и неограниченной памяти. Те же оценки верны и для задачи вычисления логического И от n аргументов.

30.2.3 Моделирование CRCW-машины с помощью EREW-машины

Мы видели, что CRCW-алгоритмы иногда работают быстрее соответствующих EREW-алгоритмов (но не наоборот: любой EREW-алгоритм может выполняться CRCW-машиной без изменений). Итак, CRCW-машина обладает большими возможно-

Рисунок 30.7 30.7. Моделирование одновременной записи на EREW-машине. (a) Шаг алгоритма в CRCW-модели с одновременной записью общего значения, когда процессоры осуществляют одновременную запись (процессоры P_0, P_2 и P_5). (b) Моделирование этого шага на EREW-машине. Вначале пара (ячейка, значение) записывается в массив A. Затем массив сортируется по первой компоненте, и записывается лишь первое из последовательности одинаковых значений.

стями, но насколько они больше? Оказывается, что если число процессоров ограничено, то можно оценить сверху преимущество CRCW-машины перед EREW-машиной:

Теорема 30.1. Для любого CRCW-алгоритма (в модели одновременной записи общего значения), использующего p процессоров, существует EREW-алгоритм для той же задачи, время работы которого больше времени работы исходного алгоритма не более чем в $O(\lg p)$ раз.

Доказательство

использует факт, рассматриваемый в разделе 30.3: существует EREW-алгоритм сортировки p элементов, который использует p процессоров и работает за время $O(\lg p)$.

Достаточно указать EREW-алгоритм, который моделирует каждый шаг CRCW-алгоритма за время $O(\lg p)$. Поскольку число процессоров в обоих случаях одно и то же, нужно лишь смоделировать одновременный доступ к памяти. Мы рассмотрим вопрос об одновременной записи, оставив одновременное чтение читателю (упр. 30.2-7).

Для моделирования одновременной записи используется дополнительный массив A размера p. Рис. 30.7 демонстрирует основную идею. Если процессор P_i (при $i = 0, 1, \ldots, n-1$) направляет для записи в ячейку с номером l_i значение x_i, то в соответствующем EREW-алгоритме пара (l_i, x_i) записывается в ячейку $A[i]$. При этом записи оказываются исключающими. Затем массив A сортируется по первой координате за время $O(\lg p)$. После этого все данные, поступившие для записи в данную ячейку, оказываются рядом.

надписи на картинке:

общая память CRCW-машины

содержимое CRCW-памяти [вместо simulated CRCW global memory]

сортировка

Затем каждый процессор P_i (при $i = 1, 2, \ldots, n-1$) сравнивает первые координаты своего и предшествующего элементов, то есть $A[i] = (l_j, x_j)$ и $A[i-1] = (l_k, x_k)$. Если $l_j \neq l_k$ или $i = 0$, то процессор P_i записывает значение x_j в ячейку l_j, а в противном случае не делает ничего. Другими словами, записывается первое из подряд идущих значений, поступивших на запись в данную *ячейку. Результат всего процесса моделирует одновременную запись, а общее время составляет $O(\lg p)$.*

Можно смоделировать и другие варианты одновременной записи (упр. 30.2-9).

Какая же модель предпочтительней — EREW или CRCW? И если CRCW, то какой из вариантов одновременной записи использовать? Сторонники модели CRCW ссылаются на то, что программы для CRCW проще и быстрее. Противники говорят, что оборудование для обеспечения одновременного доступа к памяти замедляет этот доступ, поэтому на практике выигрыш во времени является мнимым. На самом-то деле невозможно найти максимум в массиве за время $O(1)$, говорят они.

Другая точка зрения состоит в том, что модель PRAM вообще является неподходящей: процессоры должны быть соединены в сеть линиями связи, и общение должно быть возможно лишь между соседними в сети процессорами, так что структура сети является частью модели.

Впрочем, сам вопрос о "единственно правильной и подлинно научной" модели параллельных вычислений не имеет смысла. Разные аспекты реальных компьютеров отражаются разными моделями. Какой из аспектов более важен, станет ясно в будущем.

Упражнения

30.2-1. Пусть лес состоит всего лишь из одного дерева с n узлами. Покажите, что тогда существует CREW-алгоритм поиска корней со временем работы $O(1)$ независимо от глубины дерева. Покажите, что любой EREW-алгоритм даже при этом дополнительном предположении требует времени $\Omega(\lg n)$.

30.2-2. Укажите EREW-алгоритм для поиска корней со временем работы $O(\lg n)$, где n – общее количество узлов в деревьях.

30.2-3. Найдите CRCW-алгоритм, который, используя n процессоров, за время $O(1)$ вычисляет логическое ИЛИ от n аргументов.

30.2-4. Опишите эффективный CRCW-алгоритм, который умножает две булевы матрицы размера $n \times n$, используя n^3 процессоров.

30.2-5. Опишите EREW-алгоритм для умножения двух вещественных матриц размера $n \times n$ за время $O(\lg n)$, использующий n^3 процессоров. Можете ли вы найти алгоритм для CRCW-машины с записью общего значения, работающий быстрее? А для какой-либо другой модели CRCW-машины?

30.2-6. Докажите, что для любого $\varepsilon > 0$ существует CRCW-алгоритм для поиска максимального элемента в массиве размера n со временем работы $O(1)$, использующий $O(n^{1+\varepsilon})$ процессоров.*

30.2-7. Опишите алгоритм для CRCW-машины с приоритетами, позволяющий слить два отсортированных массива за время $O(1)$. Объясните, как с помощью этого алгоритма от-*

сортировать массив за время $O(\lg n)$. Является ли полученный алгоритм сортировки эффективным по затратам?

30.2-8. Объясните, как смоделировать одновременное чтение CRCW-машины с p процессорами на EREW-машине за время $O(\lg p)$ (пропущенная часть доказательства теоремы 30-1).

30.2-9. Объясните, как с помощью EREW-машины смоделировать CRCW-машину с записью комбинации значений. Для p процессоров время работы должно увеличиться не более чем в $O(\lg p)$ раз. (Указание: используйте параллельное вычисление префиксов).

30.3 Теорема Брента и эффективность по затратам

Теорема Брента даёт способ эффективно моделировать схемы из функциональных элементов с помощью параллельных машин с произвольным доступом (PRAM). С её помощью многие результаты о сортирующих сетях (глава 28) и схемах из функциональных элементов (глава 29) переносятся на модель PRAM.

Рассмотрим схему из функциональных элементов *(combinational circuit), которая состоит из* функциональных элементов *(combinational element), соединённых так, что не образуется циклов. В этом разделе предполагается, что функциональные элементы имеют любое количество входов, но ровно один выход (элемент с несколькими выходами можно заменить на несколько элементов с одним выходом). Число входов называется* входной степенью *(fan-in) элемента, а число входов, к которым подключён выход элемента — его* выходной степенью *(fan-out). Как правило, в этом разделе предполагается, что входные степени всех используемых элементов ограничены сверху. Выходные степени могут быть любыми.*

Размером *(size) схемы называется количество элементов в ней. Наибольшее число элементов на путях от входов схемы к выходу элемента называется* глубиной *(depth) этого элемента. Глубиной схемы называется наибольшая из глубин её элементов.*

Теорема 30.2 (теорема Брента)

При моделировании работы схемы глубины d и размера n с ограниченными входными степенями элементов с помощью CREW-алгоритма, использующего p процессоров, достаточно времени $O(n/p + d)$.

Доказательство.

Отведём в памяти место для входных значений схемы и для выходных значений всех функциональных элементов. Работу каждого элемента можно смоделировать с помощью одного процессора за время $O(1)$: процессор читает входные значения и вычисляет требуемую функцию, записывая результат на нужное

Рисунок 30.8 30.8. Теорема Брента. Схему размера 15 и глубины 5 можно смоделировать на CREW-машине с двумя процессорами за $9 \leqslant 15/2+5$ шагов. Элементы моделируются сверху вниз (в порядке возрастания глубины). Элементы, которые моделируются одновременно, объединены в группы (показаны серым цветом); для каждой группы указано, на каком шаге моделируются её элементы.

место; поскольку входные степени всех элементов ограничены сверху, функцию, соответствующую любому элементу, можно вычислить за время $O(1)$.

Теперь требуется указать, в какой последовательности следует моделировать работу функциональных элементов с помощью p *процессоров, чтобы обеспечить время работы* $O(n/p+d)$. *Проблема состоит в том, что выходное значение элемента можно вычислить лишь когда известны все его входные значения. (Большие выходные степени элементов не создают проблем, поскольку одновременное чтение разрешено).*

Сначала мы можем моделировать лишь элементы глубины 1 *(их входные значения известны с самого начала). После этого можно моделировать элементы глубины* 2, *затем глубины* 3 *и т. д., до глубины* d. *Заметим, что если элементы глубины* $1,\ldots,i$ *уже смоделированы, то элементы глубины* $i+1$ *можно моделировать одновременно, так как они не используют выходные значения друг друга.*

Элементы моделируются в порядке возрастания глубины, а на одной глубине — по p *штук за раз. На рис. 30.8 показано, как происходит моделирование в примере с* $p=2$.

Если число элементов глубины i *равно* n_i, *на их моделирование уйдёт не более* $(n_i/p)+1$ *шагов (единица прибавляется, если последняя группа неполная). Всего получается не более*

$$\sum_{i=1}^{d}\left(\frac{n_i}{p}+1\right)=\frac{n}{p}+d$$

шагов. Теорема доказана.

Теорема Брента справедлива и для моделирования схем на EREW-машине (при дополнительном условии, что выходные степени всех элементов также ограничены сверху):

Следствие 30.3. *Любая схема глубины* d *и размера* n, *у которой элементы имеют ограниченные сверху входные и выходные степени, может быть смоделирована с помощью EREW-алгоритма, использующего* p *процессоров, со временем работы* $O(n/p+d)$.

Доказательство. *Моделирование производится так же, как и при доказательстве теоремы Брента. Единственная разница состоит в том, что выходное значение следует скопировать в* $O(1)$ *ячеек (их количество равно максимальной выходной степени* *элемента), чтобы процессоры впоследствии могли одновременно прочесть это значение, не используя одновременное чтение из одной ячейки.*

Если выходные степени не ограничены сверху, копирование не укладывается в $O(1)$ *шагов, так что требуется одновременное чтение. Если входные степени не ограничены сверху, то в некоторых случаях (когда функциональные элементы достаточно просты) схему всё же можно смоделировать на CRCW-машине с аналогичной оценкой времени работы (упр. 30.3-1).*

В замечаниях к главе 28 указано, что существует сортирующая AKS-сеть, которая сортирует n *чисел за время* $O(\lg n)$, *используя* $O(n \lg n)$ *компараторов. Поскольку входные степени компараторов ограничены сверху, для* n *процессоров существует EREW-алгоритм сортировки со временем работы* $O((n \lg n)/n + \lg n) = O(\lg n)$ *(следствие 30.3). Этот результат был использован при доказательстве теоремы 30.1. К сожалению, множитель при логарифме столь велик, что такой алгоритм сортировки представляет лишь теоретический интерес. Однако существуют более практичные EREW-алгоритмы сортировки, например, параллельный алгоритм сортировки слиянием, принадлежащий Коулу [46].*

Пусть теперь имеется алгоритм, использующий p *процессоров, а число имеющихся процессоров* (p') *меньше* p. *Как показывает следующая теорема, моделирование возможно без увеличения общих затрат:*

Теорема 30.4. *Пусть алгоритм* A *использует* p *процессоров и имеет время работы* t. *Тогда для любого* $p' < p$ *существует алгоритм* A' *для той же задачи, использующий* p' *процессоров и имеющий время работы* $O(pt/p')$.

Доказательство. *Занумеруем шаги алгоритма* A *числами* $1, 2, \ldots, t$. *Алгоритм* A' *будет моделировать каждый шаг за время* $O(\lceil p/p' \rceil)$ *(это делается как в теореме Брента). Тогда общее время работы алгоритма* A' *будет равно* $tO(\lceil p/p' \rceil) = O(t\lceil p/p' \rceil) = O(pt/p')$, *так как* $p' < p$.

Общие затраты алгоритма A *равны* pt, *а общие затраты алгоритма* A' *равны* $O(pt/p')p' = O(pt)$, *поэтому если алгоритм* A *эффективен по затратам, то* A' *также эффективен по затратам.*

Идея доказательства проста: процессор моделирует работу нескольких процессоров в режиме "разделения времени". Напротив, "распараллеливание" — разделение работы одного процессора между несколькими — дело трудное и для каждой конкретной задачи требует специального рассмотрения.

Следующее соображение позволяет иногда решить вопрос о существовании эффективного по затратам алгоритма с данным числом процессоров. Пусть доказано, что при любом количестве *процессоров время работы алгоритма не может быть меньше t, а затраты наилучшего однопроцессорного алгоритма равны w (следовательно, затраты любого алгоритма не меньше w). Если в этой ситуации удастся найти эффективный по затратам алгоритм, использующий $p = \Theta(w/t)$ процессоров, то тем самым будут найдены эффективные алгоритмы для любого количества процессоров, при котором они существуют. Действительно, для меньшего числа процессоров такой алгоритм получается из теоремы Брента, а эффективный алгоритм для существенно большего числа процессоров p' не может существовать, поскольку время работы такого алгоритма не меньше t, а затраты не меньше $p't = \omega(pt) = \omega(w)$.*

30.3.1 Упражнения

30.3-1. *Докажите результат, аналогичный теореме Брента, для моделирования на CRCW-машине схем, содержащих элементы И и ИЛИ с любой входной степенью. (Указание: в качестве размера следует рассмотреть общее количество входов элементов).*

30.3-2. *Докажите, что существует EREW-алгоритм для параллельной обработки префиксов массива из n элементов, использующий $O(n/\lg n)$ процессоров и имеющий время работы $O(\lg n)$. Почему этот алгоритм нельзя использовать для списков?*

30.3-3. *Постройте эффективный по затратам алгоритм для умножения матрицы размера $n \times n$ на вектор размера n со временем работы $O(\lg n)$. (Указание: Постройте сначала схему из функциональных элементов.)*

30.3-4. *Постройте CRCW-алгоритм для умножения двух матриц размера $n \times n$, использующий n^2 процессоров и не менее эффективный по затратам, чем стандартный алгоритм со временем работы $\Theta(n^3)$. Будет ли он EREW-алгоритмом?*

30.3-5. *В некоторых моделях параллельных вычислений процессоры могут отключаться на некоторых шагах, так что число активных процессоров уменьшается. При подсчёте затрат на каждом шаге учитываются только активные процессоры. Докажите, что CRCW-алгоритм в такой модели со временем работы t, затратами w и любым начальным числом процессоров может быть смоделирован на EREW-машине с p процессорами за время $O((w/p + t)\lg p)$. (Основная сложность состоит в том, что заранее неизвестно, какие процессоры будут активны.)*

30.4 Эффективная параллельная обработка префиксов

В разделе 30.1.2 был рассмотрен EREW-алгоритм параллельной обработки префиксов для списка из n элементов за время $O(\lg n)$. Он использует n процессоров, поэтому затраты составляют $\Theta(n \lg n)$, что больше, чем затраты $\Theta(n)$ однопроцессорного алгоритма. Следовательно, алгоритм List-Prefix *не является эффективным по затратам.*

В этом разделе описывается вероятностный EREW-алгоритм параллельной обработки префиксов, который использует $\Theta(n/\lg n)$ процессоров и в большинстве случаев (с большой вероятностью) работает за время $O(\lg n)$. Таким образом, этот (вероятностный) алгоритм можно считать эффективным по затратам. Конструкция теоремы 30.4 позволяет указать такой алгоритм для любого числа процессоров $p = O(n/\lg n)$.

30.4.1 Рекурсивная параллельная обработка префиксов

Алгоритм Randomized-List-Prefix *использует для n-элементного списка $p = \Theta(n/\lg n)$ процессоров, так что каждый процессор отвечает за $n/p = \Theta(\lg n)$ элементов. Распределение объектов по процессорам происходит произвольным образом (не обязательно подряд) в начале работы алгоритма и более не меняется. Для удобства будем считать, что список является двунаправленным, поскольку такой список можно изготовить из однонаправленного за время $O(1)$.*

Идея алгоритма состоит в том, чтобы исключить из списка некоторые объекты (присоединив их к следующим за ними), произвести обработку префиксов для получившегося списка, а затем обработать пропущенные префиксы, учтя исключённые элементы. Один уровень рекурсии показан на рис. 30.9, а весь процесс — на рис. 30.10.

При выборе исключаемых элементов мы будем соблюдать такие правила::

1. *Нельзя одновременно исключать два элемента, за которые отвечает один и тот же процессор.*
2. *Нельзя исключать два соседних в списке объекта.*

Перед тем как описывать механизм выбора исключаемых элементов, рассмотрим подробнее сам процесс исключения. Пусть решено исключить k-й объект. Тогда $(k+1)$-й объект запоминает значение $[k, k+1] = [k, k] \otimes [k+1, k+1]$ (после чего k-й объект удаляется). Если k-й объект является последним, то он просто удаляется из списка.

После удаления процедура Randomized-List-Prefix *вызывает*

splice out = удаление
splice in = возвращение удалённых
рекурсивный вызов процедуры Randomized-List-Prefix

Рисунок 30.9 30.9. Рекурсивный вероятностный алгоритм обработки префиксов списка на примере списка из 9 объектов. (a) Чёрные объекты намечены для удаления. (Среди них нет соседних.) (b) Значение каждого удаляемого объекта соединяется со значением следующего; алгоритм рекурсивно вызывается на уменьшенном списке. (c)-(d) В результате уцелевшие элементы получают правильные значения, а удалённые объекты вычисляют свои префиксы с помощью предыдущих.

splice out удаление splice in возвращение stage 1 уровень 1 (bottom) (внутренний) (top) (внешний)

Рисунок 30.10 30.10. Уровни рекурсии для алгоритма Randomized-List-Prefix. Вначале список содержит 9 объектов. Рекурсивные вызовы осуществляются до тех пор, пока список не станет пустым.

саму себя на уменьшенном списке, если он ещё не пуст. После этого рекурсивного вызова все элементы уменьшенного списка содержат правильные значения префиксов, и остаётся лишь обработать префиксы, соответствующие исключённым элементам.

Это сделать легко: если k-й объект был исключён, то $(k-1)$-й объект остался в списке и после рекурсивного вызова содержит значение $[1, k-1]$. Остаётся лишь вычислить $[1, k] = [1, k-1] \otimes [k, k]$.

Условие 1 гарантирует, что для каждого исключённого объекта найдётся процессор, который будет производить вычисления. Условие 2 обеспечивает возможность обработать недостающий префикс, произведя всего одну операцию $\otimes$ (см. упр. 30-4.1). Итак, при выполнении требований 1 и 2 каждый шаг алгоритма требует времени $O(1)$.

30.4.2 Выбор удаляемых объектов

Как же выбирать объекты для удаления? Прежде всего, необходимо соблюдать требования 1 и 2. Кроме того, сам процесс выбора должен занимать немного времени (лучше всего $O(1)$). Наконец, следует удалять как можно больше объектов, чтобы остающийся список был как можно короче и глубина рекурсии — как можно меньше.

Укажем способ удовлетворить всем этим требованиям.

1. *Каждый процессор случайно выбирает один из подведомственных ему объектов (из числа оставшихся в списке).*
2. *Каждый процессор бросает монету и с вероятностью $1/2$ помечает выбранный им на предыдущем шаге объект (а с вероятностью $1/2$ не делает ничего).*

3. *К удалению назначаются помеченные объекты, у которых следующий объект не помечен: объект i удаляется, если он помечен, а $next[i]$ не помечен (никаким другим процессором).*

Описанные действия выполняются за время $O(1)$ и не используют одновременный доступ к памяти.

Очевидно, что требование 1 выполнено (каждый процессор выбирает не более одного элемента). Требование 2 также выполнено, так как объекты i и $next[i]$ не могут быть выбраны одновременно — если $next[i]$ выбран, то он должен быть помечен — а тогда объект i не выбирается.

30.4.3 Анализ

Поскольку шаг рекурсии занимает время $O(1)$, необходимо оценить лишь количество шагов, после которого список станет пустым. Пусть на этапе 1 описанного процесса какой-то процессор выбрал объект i. Какова вероятность того, что объект i попадёт в число удаляемых? Для этого нужно, чтобы на этапе 2 он был помечен (вероятность $1/2$) и чтобы чтобы объект $next[i]$ не был помечен (это независимое событие, имеющее вероятность не меньше $1/2$ — лишь один процессор может пометить $next[i]$, и с вероятностью $1/2$ он бездействует). Таким образом, вероятность удаления объекта i нее меньше $1/4$.

Таким образом, ситуацию можно описать так. Имеется $n/\lg n$ групп по $\lg n$ элементов в каждой (для простоты мы будем считать, что элементы делятся на группы без остатка). На каждом шаге число элементов в группе может уменьшится на 1 или остаться прежним, при этом вероятность уменьшения не менее $1/4$. Уменьшения в разных группах на одном шаге могут не быть независимыми, но уменьшения на разных шагах в одной группе независимы.

Нам надо доказать, что математическое ожидание времени полного исчезновения всех групп есть $O(\lg n)$. Анализа для одной группы тут недостаточно, поскольку даже один долго работающий процессор уже увеличивает общее время работы.

Мы покажем, что с вероятностью не менее $1-1/n$ список станет пустым через время $c\lg n$ для некоторой константы c. Полное доказательство оценки $\Theta(\lg n)$ для математического ожидания времени работы оставляется читателю (упр. 30.4-4 и 30.4-5).

Поскольку нас интересует верхняя оценка времени, когда список станет пустым, будем считать, что вероятность уменьшения данной группы на каждом шаге в точности равна $1/4$ (хотя на самом деле она может быть больше); формальное обоснование законности такого допущения даётся в упр. 6.4-8 и 6.4-9.

Рассмотрим ситуацию для одной фиксированной группы. Взяв какое-то значение c, посмотрим на вероятность того, что среди $c\lg n$ *независимых испытаний с вероятностью успеха* $1/4$ *в каждом будет меньше* $\lg n$ *успехов. (Это и будет вероятностью того, что группа останется непустой после* $c\lg n$ *испытаний.) Обозначив число успехов через* X*, оценим вероятность события* $\{X < \lg n\}$ *с помощью следствия 6.3:*

$$\mathsf{P}\{X < \lg n\} \leqslant C_{c\lg n}^{\lg n}(3/4)^{c\lg n - \lg n} \leqslant \left(\frac{ec\lg n}{\lg n}\right)^{\lg n}(3/4)^{(c-1)\lg n} = \left(ec(3/4)^{c-1}\right)^{\lg n} \leqslant (1/4)$$

(последнее неравенство верно при $c \geqslant 20$*; второе неравенство следует из (6.9)). Таким образом, вероятность того, что через* $c\lg n$ *шагов не все объекты данного процессора исключены, не превышает* $1/n^2$*.*

Всего имеется $n/\lg n$ *процессоров, поэтому вероятность " неудачи" (остаются неудалённые) одного процессора надо ещё умножить на* $n/\lg n$*, (неравенство Буля, 6.22) получается не больше*

$$\frac{n}{\lg n}\cdot\frac{1}{n^2} \leqslant \frac{1}{n}$$

Поэтому с вероятностью не менее $1 - 1/n$ *алгоритм закончит работу за время* $O(\lg n)$*.*

Константа 20 не отражает реального быстродействия алгоритма — на самом деле наша оценки довольно грубые, и среднее время значительно меньше.

30.4.4 Упражнения

30.4-1. *Нарисуйте пример, показывающий, почему нельзя исключать из списка два соседних объекта.*

30.4-2*. *Вероятностный алгоритм можно слегка изменить так, чтобы время работы в худшем случае составляло* $O(n)$*. Как? Докажите, что математическое ожидание времени работы модифицированного алгоритма равно* $O(\lg n)$*.*

30.4-3*. *Измените вероятностный алгоритм так, чтобы он использовал* $O(n/p)$ *памяти (в расчёте на один процессор) независимо от глубины рекурсии.*

30.4-4*. *Докажите, что для любого* $k \geqslant 1$ *найдётся такая константа c, что время работы алгоритма меньше* $c\lg n$ *с вероятностью* $1 - 1/n^k$ *или больше. Как зависит константа c от k?*

30.4-5*. *Докажите, используя предыдущее упражнение, что математическое ожидание времени работы вероятностного алгоритма есть* $O(\lg n)$*.*

30.5 Нарушение симметрии (детерминированный алгоритм)

Рассмотрим ситуацию, когда два процессора одновременно хотят получить доступ к объекту. При отсутствии механизма одновременного доступа к памяти один из процессоров должен получить доступ первым — но какой? Задача выбора ровно одного из двух процессоров является частным случаем задачи о нарушении симметрии *(symmetry breaking). Именно с такой задачей столкнулись Чичиков и Манилов, уступая друг другу дорогу. Подобные задачи постоянно возникают при реализации параллельных алгоритмов.*

Для решения такой задачи можно просто бросать монетку (использовать случайные числа). Например, в случае двух процессоров каждый из них бросает монетку, и доступ получает тот, у кого орёл (если два орла или две решки — бросают ещё раз). При этом симметрия будет нарушена за время $O(1)$ *(имеется в виду математическое ожидание, см. упр. 30.5-1).*

Использование случайности часто оказывается полезным — мы видели, что вероятностном алгоритме обработки префиксов (раздел 30.4) использование случайности позволяет выбрать достаточно много объектов, гарантировать, что соседние не выбраны и при этом правило выбора локально (зависит только от ситуации у двух соседних объектов).

В этом разделе рассматривается детерминированный метод нарушения симметрии. Он основан не на бросании монеты, а на использовании номеров процессоров (или адресов в памяти). Например, в случае двух процессоров можно предоставить доступ процессору с меньшим номером. При этом задача решается за постоянное время.

Мы используем ту же идею для решения более сложной задачи: как выделить (достаточно большую) часть объектов списка, если нельзя выделять два соседних объекта. Для этой задачи будет построен параллельный алгоритм (в EREW-модели), время работы которого составляет $O(\lg^* n)$*, если номера процессоров берутся из промежутка* $1..n$*. Поскольку* $\lg^* n \leqslant 5$ *при* $n \leqslant 2^{65536}$*, на практике время работы этого алгоритма можно считать постоянным (см. стр. 00).*

Алгоритм состоит из двух частей. Сначала за время $O(\lg^* n)$ *мы находим "6-раскраску" списка. Затем (за время* $O(1)$*) мы получаем из неё "максимальное независимое подмножество" списка; оно содержит не менее трети объектов списка и не содержит соседних объектов.*

30.5.1 Раскраски и максимальные независимые множества

Раскраской *(coloring) неориентированного графа* $G = (V, E)$ *называется отображение* $C : V \to \mathbb{N}$, *для которого* $C(u) \neq C(v)$ *при* $(u, v) \in E$. *Если называть* $C(v)$ *цветом вершины* v, *то можно сказать, что концы любого ребра должны иметь разные цвета. Мы ищем 6-раскраску списка, то есть хотим поставить в соответствие каждой вершине некоторый цвет из множества* $\{0, 1, 2, 3, 4, 5\}$, *причём так, что соседние в списке вершины имеют разные цвета.*

Такая раскраска существует; более того, ясно, что достаточно двух цветов. В самом деле, можно покрасить все чётные вершины в один цвет, а все нечётные — в другой. Но находясь в середине списка, не так просто понять, чётная вершина или нечётная — для этого нужно отсчитать её номер от начала списка, что можно сделать за время $O(\lg n)$ *(раздел 30.1). Разрешив больше цветов (как мы увидим, достаточно* 6*), раскраску можно построить быстрее — за время* $O(\lg^* n)$, *при этом алгоритм остаётся детерминированным.*

Множество $V' \subseteq V$ *вершин графа* $G = (V, E)$ *назовём* независимым множеством *(independent set), если никакие две вершины из* V' *не соединены ребром. Независимое множество* V' *называется* максимальным *(maximal independent set, MIS), если при добавлении любой вершины из* $V \setminus V'$ *оно перестаёт быть независимым.*

Не следует смешивать задачу нахождения такого множества с гораздо более сложной задачей нахождения независимого множества наибольшей мощности (каковое, конечно, является максимальным — но обратное неверно). Для произвольного графа последняя задача является NP-полной (см. главу 36, задача 36.1). Для n*-элементного списка независимое множество максимальной мощности состоит из* $\lceil n/2 \rceil$ *объектов — достаточно взять объекты через один, начиная с первого. Это множество, как и соответствующая 2-раскраска, может быть найде)о с помощью параллельной обработки префиксов за время* $O(\lg n)$.

Заметьте, что в случае списка любое максимальное множество вершин содержит не меньше $n/3$ *элементов. Действительно, из любых трёх подряд идущих объектов хотя бы один должен входить в множество — иначе средний из этих трёх можно добавить, сохранив независимость.*

Ниже будет показано, как за время $O(1)$ *построить максимальное независимое множество по* $O(1)$*-раскраске.*

Рисунок 30.11 30.11. Построение 6-раскраски и соответствующего максимального независимого множества. Алгоритм использует n процессоров и работает за время $O(\lg^* n)$. Вначале список из $n = 20$ объектов раскрашен в 20 цветов (что требует пяти битов). За два шага эта раскраска сводится к 6-раскраске. Элементы MIS показаны чёрным цветом.

30.5.2 Вычисление 6-раскраски

Укажем алгоритм, находящий 6-раскраску списка. Мы считаем, что за каждым элементом x списка закреплён процессор процессор, номер которого (известный процессору) есть $P(x) \in \{0, 1, \ldots, n-1\}$.

Алгоритм последовательно строит раскраски $C_0, C_1, \ldots, C_m$, постепенно уменьшая число цветов. Начальная раскраска C_0 является n-раскраской, а C_m – 6-раскраской. На k-ом шаге по раскраске C_k вычисляется раскраска C_{k+1}. Число шагов m есть $O(\lg^* n)$.

Начальная раскраска C_0 тривиальна: $C_0[x] = P(x)$ (цвет вершины есть номер соответствующего ей процессора). Поскольку номера процессоров различны, это отображение действительно является раскраской. (Заметьте, что каждый цвет кодируется $\lceil \lg n \rceil$ битами и может храниться в памяти, занимая не слишком много места.)

Опишем теперь процесс построения раскраски C_{k+1} по C_k (см. рис. 30.11). Мы считаем, что каждый цвет кодируется битовой строкой фиксированной (для данной раскраски) длины. Переход от C_k к C_{k+1} уменьшает эту длину.

Итак, пусть цвета раскраски C_k записываются r битами и соседние объекты x и $next[x]$ имеют цвета $C_k[x] = a$ и $C_k[next[i]] = b$, где $a = \langle a_{r-1}, a_{r-2}, \ldots, a_0 \rangle$, $b = \langle b_{r-1}, b_{r-2}, \ldots, b_0 \rangle$. Цвета эти различны, поэтому $a_i \neq b_i$ для некоторого i от 0 до $r-1$. Мы объявим пару $\langle i, a_i \rangle$ цветом элемента x в новой раскраске (закодировав её последовательностью битов). Цветом последнего элемента в новой раскраске мы будем считать пару $\langle 0, a_0 \rangle$, если $\langle a_{r-1}, a_{r-2}, \ldots, a_0 \rangle$ было его цветом в старой раскраске.

Остаётся понять, почему в новой раскраске цвета соседних вершин будут различны и сколько битов нужно для кодирования её цветов. Начнём с первого; рассмотрим два соседних элемента x и $y = next[x]$ в нашем списке. Раньше они имели цвета $a = \langle a_{r-1}, a_{r-2}, \ldots, a_0 \rangle$, $b = \langle b_{r-1}, b_{r-2}, \ldots, b_0 \rangle$, которые различались в бите i, и новый цвет x есть пара $\langle i, a_i \rangle$, причём $a_i \neq b_i$. Пусть новый цвет y есть пара $\langle j, b_j \rangle$. Видно, что пары эти различны: если различны их первые члены, то и доказывать нечего, если же первые члены совпадают ($i = j$), то различны вторые b_j равно b_i и не равно a_i.

Теперь о количестве битов: если на k-ом шаге цвета записы-

вались r битами, то на $(k+1)$-ом шаге они будут записываться всего $\lceil \lg r \rceil + 1$ битами. Если $r \geqslant 4$, то при этом число битов уменьшается. Если же $r = 3$, то два цвета могут различаться в разрядах 0, 1 или 2, поэтому на следующем шаге номер любого цвета будет начинаться с $\langle 00 \rangle$, $\langle 01 \rangle$ или $\langle 10 \rangle$ и оканчиваться нулём либо единицей. При этом из восьми цветов получается не более шести, так что мы приходит к 6-раскраске.

Если предположить, что операция поиска может найти нужный индекс и совершить левый сдвиг за время $O(1)$, то каждый шаг занимает время $O(1)$. Алгоритм можно реализовать на EREW-машине, так как каждому процессору необходим доступ лишь к двум объектам — x и $next[x]$.

Покажем теперь, что число шагов до получения 6-раскраски составляет $O(\lg^* n)$. Напомним, что $\lg^* n$ определялось как число применений к n функции $\lg$, после которых результат будет не больше 1. Более точно, если через $\lg^{(i)} n$ обозначить i-кратное применение логарифма, то

$$\lg^* n = \min \left\{ i \geqslant 0 : \lg^{(i)} n \leqslant 1 \right\}$$

В нашем случае число битов на каждом шаге тоже логарифмируется, но затем округляется (с избытком) и увеличивается на 1. Проверим, что всё равно число шагов есть $O(\lg^* n)$.

Пусть r_i – число битов, которыми записываются цвета перед i-м шагом. Мы докажем по индукции, что $r_i \leqslant \lceil \lg^{(i)} n \rceil + 2$ при $\lceil \lg^{(i)} n \rceil \geqslant 2$ Изначально $r_1 \leqslant \lceil \lg n \rceil$. Пусть утверждение верно для $(i-1)$-го шага. Поскольку $r_i = \lceil \lg r_{i-1} \rceil + 1$, имеем

$$\begin{aligned} r_i = \lceil \lg r_{i-1} \rceil + 1 &\leqslant \lceil \lg(\lceil \lg^{(i-1)} n \rceil + 2) \rceil + 1 \leqslant \lceil \lg(\lg^{(i-1)} n + 3) \rceil + 1 \leqslant \\ &\lceil \lg(2 \lg^{(i-1)} n) \rceil + 1 = \lceil \lg(\lg^{(i-1)} n) + 1 \rceil + 1 = \lceil \lg^{(i)} n \rceil + 2 \end{aligned}$$

Четвёртое неравенство получается так: если $\lceil \lg^{(i)} n \rceil \geqslant 2$, то $\lceil \lg^{(i-1)} n \rceil \geqslant 3$, так что увеличение на 3 можно заменить умножением на 2. Поэтому для $m = \lg^* n - 1$ имеем $r_m \leqslant \lceil \lg^{(m-1)} \rceil + 2 \leqslant 4$, так как $\lg^{(m+1)} n \leqslant 1$ и $\lg^{(m)} n \leqslant 2$ по определению $\lg^* n$. Значит, $r_{m+1} \leqslant 3$, и ещё через шаг процесс закончится. Таким образом, общее количество шагов составляет $O(\lg^* n)$.

30.5.3 Получение максимального независимого множества из 6-раскраски

Пусть дан список из n объектов и его раскраска C в c цветов. Опишем EREW-алгоритм, который позволяет найти по этой раскраске максимальное независимое множество (MIS) за время $O(c)$. В частности, зная 6-раскраску, можно найти максимальное независимое множество за время $O(1)$.

Идея алгоритма показана на рис. 30.11 (шесть правых колонок). Для каждого объекта x соответствующий процессор хранит бит $alive[x]$. При этом $alive[x] =$ TRUE означает, что элемент x ещё имеет шанс попасть в MIS. Вначале $alive[x] =$ TRUE для всех объектов x.

Алгоритм на каждом шаге рассматривает элементы одного цвета. Пусть текущий цвет равен i. Каждый процессор проверяет для своего элемента x условия $C[x] = i$ и $alive[x] =$ TRUE. Если оба условия выполнены, то элемент x включается в MIS, а alive-биты соседних с ним элементов (следующего и предыдущего) устанавливаются равными FALSE *(эти элементы нельзя включать в MIS). После с шагов каждый объект либо включён в MIS, либо имеет alive-бит, равный* FALSE.

Покажем, что полученное множество действительно является независимым и максимальным. Предположим, что в множество попали два соседних объекта. В силу свойств раскраски они имеют разные цвета, поэтому включены на разных шагах. Но включённый первым объект должен был установить alive-бит второго в положение FALSE, *и второй элемент уже не мог быть включён.*

Максимальность множества очевидна. Действительно, если элемент x не включён в множество, то $alive[x] =$ FALSE, поэтому в множество включён один из его соседей. Поэтому при добавлении элемента x множество перестанет быть независимым.

Каждый шаг алгоритма требует времени $O(1)$. Алгоритм может быть реализован на EREW-машине, поскольку каждый процессор обращается только к трём объектам – своему и его соседям. Вместе с алгоритмом поиска 6-раскраски этот алгоритм даёт возможность найти максимальное независимое множество (и тем самым решить задачу о нарушении симметрии) за время $O(\lg^* n)$ с помощью детерминированного алгоритма.

30.5.4 Упражнения

30.5-1 Докажите, что в примере с выбором одного из двух процессоров с помощью бросания монетки математическое ожидание требуемого времени конечно.

30.5-2 Пусть дана 6-раскраска списка из n элементов. Найдите EREW-алгоритм, который за время $O(1)$ строит 3-раскраску того же списка, используя n процессоров.

30.5-3 Пусть дерево с n узлами задано следующим образом: каждый узел, кроме корня, имеет указатель на родителя. Найдите CREW-алгоритм, который строит $O(1)$-раскраску этого дерева за время $O(\lg^* n)$.

30.5-4. Придумайте эффективный алгоритм поиска $O(1)$-*

раскраски графа степени 3 и оцените время его работы.

30.5-5. Назовём* k-разделённым *подмножеством списка (k-ruling set) множество его объектов, которое не содержит соседних объектов и не имеет дырок длины больше* k *(то есть его дополнение не содержит более* k *подряд идущих объектов). Например, любое максимальное независимое множество будет* 2*-разделённым. Как найти* $O(\lg n)$*-разделённое подмножество в* n*-элементном списке за время* $O(1)$*, используя* n *процессоров? Тот же вопрос для* $O(\lg\lg n)$*-множества.*

30.5-6 Придумайте алгоритм поиска* 6*-раскраски* n*-элементного списка за время* $O(\lg(\lg^* n))$*, предполагая, что каждый процессор может хранить заранее вычисленную таблицу размера* $O(\lg n)$*. (Указание. От скольких значений зависит конечный цвет объекта в построенной нами 6-раскраске?)*

30.6 Задачи

30-1 Обработка префиксов на отрезках

При обработке префиксов на отрезках *(segmented prefix computation), как и при обычной обработке префиксов, фиксирована бинарная ассоциативная операция* $\otimes$ *на множестве* S*. По данной последовательности* $x = \langle x_1, x_2, \ldots, x_n \rangle$*,* $x_i \in S$ *и* последовательности границ *(segment sequence)* $b = \langle b_1, b_2, \ldots, b_n \rangle$*, где* $b_i \in \{0, 1\}$*,* $b_1 = 1$*, требуется вычислить последовательность* $\langle y_1, y_2, \ldots, y_n \rangle$*,* $y_i \in S$*, которая строится так: каждая единица в последовательности* b *означает начало нового отрезка, и на каждом отрезке независимо производится обработка префиксов (см. рис. 30.12).*

a. Определим на множестве пар $\{0, 1\} \times S$ *новую операцию* $\hat{\otimes}$*:*

$$(a, z)\hat{\otimes}(a', z') = \begin{cases} (a, z \otimes z') & \text{если } a' = 0 \\ (1, z') & \text{если } a' = 1 \end{cases}$$

Докажите, что эта операция ассоциативна.

b). *Найдите EREW-алгоритм для обработки префиксов на отрезках за время* $O(\lg n)$*.*

c). *Найдите EREW-алгоритм, сортирующий список из* n k*-разрядных чисел за время* $O(k \lg n)$*.*

30-2 Поиск максимума из n чисел с помощью n процессоров

Здесь описан CRCW-алгоритм нахождения максимума из n *чисел, использующий* $p = n$ *процессоров.*

Рисунок 30.12 30.12. Обработка префиксов на отрезках (операция — сложение). Показаны входные последовательности x и b и выходная последовательность y. В этом примере имеется 5 отрезков.

a. Докажите, что если $m \leqslant p/2$, то нахождение максимума из m чисел можно за время $O(1)$ свести к нахождению максимума из не более чем m^2/p чисел, используя CRCW-машину с p процессорами.

b. Пусть вначале имеется $m = \lfloor p/2 \rfloor$ чисел. Сколько их останется после k применений конструкции пункта (a)?

c. Постройте CRCW-алгоритм поиска максимума из n чисел со временем работы $O(\lg\lg n)$, использующий $p = n$ процессоров.

30-3 Связные компоненты

В этой задаче строится алгоритм для поиска связных компонент неориентированного графа $G = (V, E)$. Алгоритм предназначен для CRCW-машины (в которой может записаться любое из одновременно записываемых значений) с $|V + E|$ процессорами. Для каждой вершины v мы храним указатель $p[v]$. Вначале $p[v] = v$ для всех вершин v. В конце $p[v] = p[u]$ в том и только в том случае, когда u и v лежат в одной связной компоненте. Во время работы алгоритма указатели образуют несколько *деревьев ссылок* (pointer trees). Дерево ссылок называется *звездой* (star), если $p[u] = p[v]$ для любых двух его вершин u и v.

Предполагается, что каждое ребро в графе продублировано, то есть вместе с парой (u, v) множество E содержит пару (v, u). Алгоритм использует две основных операции: Hook и Jump, а также процедуру Star, которая делает $star[v]$ равным TRUE, если вершина v принадлежит звезде.

```
Hook(G)
Star(G)
for(для) каждого ребра $(u, v) \in E[G]$
    do if $star[u]$ и $p[u] > p[v]$
          then $p[p[u]] \leftarrow p[v]$
Star(G)
for(для) каждого ребра $(u, v) \in E[G]$
    do if $star[u]$ и $p[u] \neq p[v]$
          then $p[p[u]] \leftarrow p[v]$
```

```
Jump(G)
for(для) каждой вершины $v \in V[G]$
    do $p[v] \leftarrow p[p[v]]$
```

Алгоритм работает следующим образом: вначале выполняется Hook, а затем чередуются Hook, Jump, Hook, Jump и так далее, пока при выполнении процедуры Jump не окажется, что ни один указатель не меняется. Заметьте, что вначале дважды выполняется процедура Hook.

a. Напишите процедуру Star(G).

b. Докажите, что указатели p образуют в любой момент несколько деревьев, причём корни находятся в тех вершинах, которые указывают на себя. Докажите, что если $p[u] = p[v]$, то u и

v лежат в одной связной компоненте.

c. Докажите, что алгоритм работает правильно, то есть останавливается и в момент остановки $p[u] = p[v]$ тогда и только тогда, когда u и v принадлежат одной связной компоненте.

Оценим время работы алгоритма. Рассмотрим какую-нибудь связную компоненту C, содержащую не меньше двух вершин. Пусть в некоторый момент работы алгоритма компонента C состоит из нескольких деревьев $\{T_i\}$. Определим потенциал компоненты C как

$$\Phi(C) = \sum_{T_i} height(T_i)$$

где $height(T_i)$ – высота дерева T_i. Наша цель – показать, что каждая пара операций Hook–Jump *уменьшает $\Phi(C)$ в некоторое фиксированное число раз (или ещё сильнее).*

d. Докажите, что после первого вызова процедуры Hook *нет деревьев высоты* 0 *и что $\Phi(C) \leqslant |V|$.*

e. Докажите, что последующие вызовы процедуры Hook *не увеличивают $\Phi(C)$.*

f. Докажите, что после любого вызова процедуры Hook*, кроме первого, среди деревьев ссылок нет звёзд (за исключением связных компонент, все вершины которых уже объединены в одну звезду).*

g. Докажите, что если деревья ссылок для данной связной компоненты C ещё не превратились в одну звезду, то после вызова процедуры Jump *потенциал $\Phi(C)$ уменьшается по крайней мере в полтора раза. Каков наихудший случай (когда потенциал уменьшается меньше всего)?*

h. Выведите из предыдущего, что время работы алгоритма составляет $O(\lg |V|)$.

30-4 Транспонирование изображения

Видеопамять можно рассматривать как матрицу битов M размера $p \times p$ (считаем, что точки бывают чёрными и белыми). При этом на дисплее видна её часть — верхняя левая подматрица размера $n \times n$. Определим операцию переноса блока битов *(Block Transfer of bits,* BitBLT*). Процедура* BitBLT*$(r_1, c_1, r_2, c_2, nr, nc, *)$ выполняет присваивания*

$$M[r_2 + i, c_2 + j] \leftarrow M[r_2 + i, c_2 + j] * M[r_1 + i, c_1 + j]$$

для всех $i = 0, 1, \ldots, nr - 1$, $j = 0, 1, \ldots, nc - 1$. Здесь $*$ обозначает любую из 16 *бинарных логических операций.*

Пусть требуется транспонировать видимую часть изображения: $M[i, j] \leftarrow M[j, i]$. Будем предполагать, что время копирования битов меньше, чем время переноса блока битов, так что мы оцениванием именно число операций BitBLT.

Докажите, что можно транспонировать изображение, вызвав процедуру BitBLT *всего лишь $O(\lg n)$ раз. Предполагается, что*

p существенно больше, чем n, так что имеется большое рабочее пространство, где можно сохранять промежуточные результаты. Какая часть этого пространства используется? (Указание: используйте подход "разделяй и властвуй", запуская процедуру BitBLT *с операцией И).*

30.7 Комментарии

Параллельные алгоритмы для комбинаторных задач описывают Акл [9], Карп и Рамачандра [118] и Лейтон [135]. Различные модели параллельных вычислений описали Хванг и Бриггс [109], а также де Гроот [110].

Начало теории параллельных вычислений относится к концу 1940-х годов XX века, когда Дж. фон Нейман [38] ввёл понятие клеточного автомата, который по сути является двумерным массивом процессоров с конечным числом состояний, расположенных в узлах клетчатой бумаги. сетке. Модель PRAM ввели Форчун и Вайли [73] (похожие модели предлагались многими авторами и раньше).

Метод переходов по указателям был предложен Вайли [204]. Параллельная обработка префиксов была рассмотрена в работе Офмана [152] (в контексте сложения с предвычислением переносов). Метод эйлерова цикла предложили Тарьян и Вишкин [191].

Соотношения между числом процессоров и временем для задачи нахождения максимума n элементов изучал Валиантом [193]; он же доказал, что для этой задачи не существует эффективного по затратам алгоритма со временем работы $O(1)$. Кук, Дворк и Райшук [50] доказали, что вычисление максимума на CREW-машине требует времени $\Omega(\lg n)$. Метод моделирования CRCW-машины с помощью EREW-машины предложил Вишкин [195].

Теорему 30.2 доказал Брент [34]. Эффективный эффективный алгоритм для нахождения номеров в списке предложили Андерсон и Миллер [11]. Они предложили также эффективный детерминированный алгоритм для этой задачи [10]. Детерминированный алгоритм нарушения симметрии взят из работы Гольдберга и Плоткина [84]; он основан на похожем алгоритме с тем же временем работы, который придумали Коул и Вишкин [47].

31 Матрицы и действия с ними

Матрицы часто встречаются в научных расчётах, поэтому важно уметь эффективно с ними работать. Эта глава содержит краткое введение в теорию матриц и операций над ними. Особое внимание уделяется умножению матриц и решению систем линейных уравнений.

В разделе 31.1 мы введём основные понятия и обозначения. В разделе 31.2 излагается алгоритм Штрассена, позволяющий умножить две матрицы размера $n \times n$ *за время* $\Theta(n^{\lg 7}) = O(n^{2.81})$. *(Появление алгоритма умножения матриц, работающего быстрее, чем стандартный, было в своё время большой неожиданностью.) В разделе 31.3 мы введём понятия квазикольца, кольца и поля, а затем сформулируем допущения, необходимые для правильной работы алгоритма Штрассена. Этот раздел содержит также асимптотически быстрый алгоритм перемножения булевых матриц, основанный на алгоритме Штрассена. В разделе 31.4 рассказывается, как решать системы линейных уравнений с помощью так называемого LUP-разложения. В разделе 31.5 мы обсудим связь между задачей умножения матриц и задачей обращения матрицы. Наконец, в разделе 31.6 мы рассмотрим симметрические положительно определённые матрицы и покажем, как с их помощью можно решать переопределённые системы линейных уравнений методом наименьших квадратов.*

31.1 Матрицы и их свойства

В этом разделе мы напомним основные понятия теории матриц, используемые в дальнейшем.

Матрицы и векторы.

Матрицей *(matrix) называется прямоугольная таблица чисел.*

Например,

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} = \\ = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \tag{31.1}$$

является 2×3 *матрицей* $A = (a_{ij})$, *в которой на пересечении* i*-й строки и* j*-го столбца стоит элемент* a_{ij} *(*$i = 1, 2$ *и* $j = 1, 2, 3$*). Мы будем обозначать матрицы большими буквами, а их элементы — соответствующими маленькими буквами с нижними индексами. Множество всех* $(m \times n)$*-матриц с вещественными элементами обозначается* $\mathbb{R}^{m \times n}$. *В общем случае, множество матриц размера* $m \times n$, *элементы которых берутся из множества* S, *обозначается* $S^{m \times n}$.

При транспонировании *матрицы* A *её строки становятся столбцами и наоборот. Матрица, получаемая из* A *транспонированием, обозначается* A^T *(the transpoose of* A*). Например, для матрицы* A *из (31.1)*

$$A^T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

Вектором *(vector) называется одномерный массив чисел. Например,*

$$x = \begin{pmatrix} 2 \\ 3 \\ 5 \end{pmatrix} \tag{31.2}$$

является вектором из трёх элементов. Мы будем обозначать векторы маленькими буквами. Для i*–го элемента вектора* x, *состоящего из* n *элементов, применяется обозначение* x_i *(*$i = 1, 2, ..., n$*). Стандартной формой вектора мы будем считать* вектор-столбец *(то есть* $n \times 1$*-матрицу); при его транспонировании получается* вектор-строка*:*

$$x^T = \begin{pmatrix} 2 & 3 & 5 \end{pmatrix}.$$

Вектор, i*-й элемент которого равен 1, а все остальные элементы равны 0, называют* единичным вектором *(unit vector) и обозначают* e_i. *Количество элементов единичного вектора обычно определяется из контекста.*

Нулевой матрицей *(zero matrix) называется матрица, все элементы которой равны* 0. *Такая матрица обычно обозначается* 0. *Что понимать под этим обозначением — число* 0 *или нулевую матрицу — обычно ясно из контекста; если имеется в виду матрица, то размер её тоже определяется из контекста.*

Часто встречаются квадратные матрицы *(square matrices) — матрицы размера $n \times n$. Некоторые их виды мы отметим особо.*

1. У диагональной матрицы *(diagonal matrix) все внедиагональные элементы равны нулю ($a_{ij} = 0$ при $i \neq j$), поэтому она может быть задана перечислением элементов, стоящих на диагонали.*

$$diag(a_{11}, a_{22}, \ldots, a_{nn}) = \begin{pmatrix} a_{11} & 0 & \ldots & 0 \\ 0 & a_{22} & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & a_{nn} \end{pmatrix}.$$

2. Единичной матрицей *(identity matrix) называется диагональная матрица, диагональ которой заполнена единицами:*

$$I_n = diag(1, 1, ..., 1) = \begin{pmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \end{pmatrix}.$$

Иногда индекс n при букве I опускается; размер матрицы в этом случае определяется из контекста. Столбцами единичной матрицы служат векторы $e_1, e_2, \ldots, e_n$.

3. У трёхдиагональной матрицы *(tridiagonal matrix) ненулевые элементы могут появляться на главной диагонали ($t_{i,i}$ при $i = 1, 2, \ldots, n$), прямо над ней ($t_{i,i+1}$ при $i = 1, 2, \ldots, n-1$), или прямо под ней ($t_{i+1,i}$ при $i = 1, 2, \ldots, n-1$). Все остальные элементы равны нулю ($t_{ij} = 0$ при $|i - j| > 1$):*

$$T = \begin{pmatrix} t_{11} & t_{12} & 0 & 0 & \ldots & 0 & 0 & 0 \\ t_{21} & t_{22} & t_{23} & 0 & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & t_{n-2,n-2} & t_{n-2,n-1} & 0 \\ 0 & 0 & 0 & 0 & \ldots & t_{n-1,n-2} & t_{n-1,n-1} & t_{n-1,n} \\ 0 & 0 & 0 & 0 & \ldots & 0 & t_{n,n-1} & t_{n,n} \end{pmatrix}.$$

4. У верхне-треугольной матрицы *(upper-triangular matrix) все элементы под главной диагональю равны нулю ($u_{ij} = 0$ при $i > j$):*

$$U = \begin{pmatrix} u_{11} & u_{12} & \ldots & u_{1n} \\ 0 & u_{22} & \ldots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & u_{nn} \end{pmatrix}.$$

5. У нижне-треугольной матрицы *(lower-triangular matrix) все* *элементы над главной диагональю равны нулю ($l_{ij} = 0$ при $i < j$):*

$$L = \begin{pmatrix} l_{11} & 0 & \dots & 0 \\ l_{12} & l_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \dots & l_{nn} \end{pmatrix}.$$

6. Матрица перестановки *(permutation matrix) имеет в точности одну единицу в каждой строке и каждом столбце; на всех прочих местах у неё стоят нули. Пример матрицы перестановки:*

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

Умножение вектора x на матрицу перестановки приводит к перестановке его элементов.

7. Симметрическая матрица *(symmetric matrix) удовлетворяет условию $A = A^T$. Например, матрица*

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 6 & 4 \\ 3 & 4 & 5 \end{pmatrix}$$

является симметрической.

Действия с матрицами.

Элементами матрицы или вектора служат элементы некоторой числовой системы (действительные числа, комплексные числа, остатки по модулю простого числа). Операции сложения и умножения в этой числовой системе можно распространить на матрицы с элементами из неё.

Определим сложение матриц *(matrix addition) следующим образом. Пусть даны $(m \times n)$-матрицы $A = (a_{ij})$ и $B = (b_{ij})$. Назовём их суммой $(m \times n)$-матрицу $C = (c_{ij}) = A + B$ с элементами*

$$c_{ij} = a_{ij} + b_{ij},$$

где $i = 1, 2, \dots, m$ и $j = 1, 2, \dots, n$. Другими словами, сложение матриц осуществляется покомпонентно. Нулевая матрица является нейтральным элементом для операции сложения матриц:

$$\begin{aligned} A + 0 &= A \\ &= 0 + A \end{aligned}$$

Пусть λ – число, а $A = (a_{ij})$ — матрица. Можно умножить матрицу A на число λ, *умножив каждый элемент A на λ. Результат* *умножения -матрица* $\lambda A = (\lambda a_{ij})$ *(scalar multiple of A). Особо отметим матрицу* $-A = (-1) \cdot A$, *называемую* противоположной *к A матрицей (negative of a matrix A). Элемент с индексами ij в матрице* $-A$ *равен* $-a_{ij}$, *поэтому*

$$\begin{aligned} A + (-A) &= 0 \\ &= (-A) + A \end{aligned}$$

Вычитание матрицы *(matrix subtraction) мы теперь можем определить как прибавление противоположной матрицы:* $A - B = A + (-B)$.

Умножение *матрицы A на матрицу B (matrix multiplication) осуществимо, лишь если они имеют согласованные размеры, то есть число столбцов A совпадает с числом строк B. (Запись AB подразумевает, что матрицы A и B имеют согласованные размеры.) Если* $A = (a_{ij})$ — $(m \times n)$*-матрица, а* $B = (b_{ij})$ — $(n \times p)$*-матрица, то их произведением* $C = AB$ *называется* $(m \times p)$*-матрица* $C = (c_{ij})$, *в которой*

$$c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk} \tag{31.3}$$

для $i = 1, 2, \ldots, m$ *и* $k = 1, 2, \ldots, p$. *Именно эту формулу реализует процедура* MATRIX-MULTIPLY *из раздела 26.1, умножающая квадратные матрицы* $(m = n = p)$. *Чтобы вычислить произведение двух* $(n \times n)$ *матриц,* MATRIX-MULTIPLY *выполняет* n^3 *умножений и* $n^2(n-1)$ *сложений, так что время её работы есть* $\Theta(n^3)$.

Матрицы обладают многими (хотя и не всеми) свойствами чисел. Единичная матрица является нейтральным элементом для умножения: для любой $(m \times n)$*-матрицы A:*

$$I_m A = A I_n = A$$

Умножение на нулевую матрицу даёт нулевую матрицу:

$$A0 = 0.$$

Умножение матриц ассоциативно:

$$A(BC) = (AB)C \tag{31.4}$$

для любых матриц A, B и C согласованных размеров. Умножение матриц дистрибутивно относительно сложения:

$$\begin{aligned} A(B+C) &= AB + AC \\ (B+C)D &= BD + CD \end{aligned} \tag{31.5}$$

При $n \neq 1$ *умножение* $n \times n$ *матриц, вообще говоря, не коммутативно. Напрмер, для* $A = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$ *и* $B = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$ *имеем*

$$AB = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix},$$

но

$$BA = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$

Считая вектор-столбец $(n \times 1)$*-матрицей, а вектор-строку —* $(1 \times n)$*-матрицу, мы можем перемножать векторы и матрицы. Если* A *—* $m \times n$ *матрица, а* x *— вектор из* n *компонент, то произведение* Ax *есть вектор из* m *компонент. Если* x *и* y *- векторы из* n *компонент, то произведение*

$$x^T y = \sum_{i=1}^{n} x_i y_i$$

представляет собой число ((1×1)*−матрицу), называемое* скалярным произведением *(inner product)* x *и* y*. Матрица* $Z = xy^T$ *размера* $n \times n$ *с элементами* $z_{ij} = x_i z_j$ *называется* тензорным произведением *(outer product) этих же векторов.* Евклидова норма *(euclidean norm)* $\|x\|$ *вектора* x *определяется равенством*

$$\begin{aligned} \|x\| &= (x_1^2 + x_2^2 + \cdots + x_n^2)^{1/2} \\ &= (x^T x)^{1/2}. \end{aligned}$$

Норма вектора x *— это его длина в* n*-мерном евклидовом пространстве.*

Обратная матрица, ранг и детерминант.

Матрицей, обратной к $(n \times n)$*−матрице* A *(inverse of* A*) называется матрица* A^{-1}*, для которой* $AA^{-1} = I_n = A^{-1}A$*. (если таковая существует). Например,*

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{-1} = \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}.$$

Многие ненулевые $n \times n$ *матрицы не имеют обратных. Матрицы, не имеющие обратных, называются* необратимыми *(noninvertible) или* вырожденными *(singular). Пример ненулевой вырожденной матрицы:*

$$\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}.$$

Матрица, имеющая обратную, называется обратимой *(invertible) или* невырожденной *(nonsingular). Если обратная матрица существует, то она единственна (см.упражнение 31.1-4). Если* $n \times n$

матрицы A и B обратимы, то

$$(BA)^{-1} = A^{-1}B^{-1}. \qquad (36.1)$$

Операция обращения матрицы перестановочна с операцией транспонирования:

$$\left(A^{-1}\right)^T = \left(A^T\right)^{-1}.$$

Говорят, что векторы $x_1, x_2, \ldots, x_n$ линейно зависимы *(linearly dependent), если найдётся набор коэффициентов $c_1, c_2, \ldots, c_n$, не все из которых равны нулю, для которого $c_1x_1 + c_2x_2 + \cdots + c_nx_n = 0$. Например, векторы $\begin{pmatrix}1 & 2 & 3\end{pmatrix}^T$, $\begin{pmatrix}2 & 6 & 4\end{pmatrix}^T$, и $\begin{pmatrix}4 & 11 & 9\end{pmatrix}^T$ зависимы, поскольку $2x_1 + 3x_2 - 2x_3 = 0$. Векторы, не являющиеся линейно зависимыми, называются* линейно независимыми *(linearly independent). Таковы, например, столбцы единичной матрицы.*

Столбцовым рангом *(column rank) ненулевой $m \times n$-матрицы A называется наибольшее число линейно независимых столбцов A.* Строчным рангом *(row rank) называется наибольшее число линейно независимых строк. Для любой матрицы A эти два числа рангов совпадают, так что их общее значение называется просто* рангом *(rank) A. Ранг $(m \times n)$-матрицы представляет собой целое число в пределах от 0 до $\min(n, m)$. (Ранг нулевой матрицы равен 0, а ранг единичной матрицы I_n равен n.) Приведём эквивалентное определение, иногда более удобное: рангом ненулевой $(m \times n)$-матрицы A называется наименьшее число r, для которого найдутся матрицы B и C размеров $m \times r$ и $r \times n$ соответственно, для которых*

$$A = BC.$$

Квадратная $n \times n$ матрица ранга n называется матрицей полного ранга *(has full rank). Основное свойство рангов таково:*

Теорема 31.1

Квадратная матрица имеет полный ранг тогда и только тогда, когда она невырождена.

Матрица размера $m \times n$, имеющая ранг n, называется матрицей полного столбцового ранга *(has full column rank).*

Ненулевой вектор x, для которого $Ax = 0$, называется аннулирующим вектором *(null vector) матрицы A. Следующая теорема (доказательство которой оставляется читателю в качестве упр. 31.1-8) и её следствие устанавливают связь между существованием аннулирующего вектора, вырожденностью и величиной столбцового ранга.*

Теорема 31.2

Матрица имеет полный столбцовый ранг тогда и только тогда, когда для неё не существует аннулирующего вектора.

Следствие 31.3

Квадратная матрица вырождена тогда и только тогда, когда для неё найдётся аннулирующий вектор.

Минором *элемента a_{ij} матрицы A размера $n\times n$ (ijth minor) называется $(n-1)\times(n-1)$-матрица $A_{[ij]}$, получаемая вычёркиванием i–й строки и j–го столбца в матрице A. Теперь* определитель *(determinant) матрицы A задаётся такой рекурсивной формулой:*

$$\det(A) = \begin{cases} a_{11} & \text{если} \quad n = 1, \\ a_{11}\det(A_{[ij]}) - a_{12}\det(A_{[12]}) + & \\ + \cdots + (-1)^{n+1}a_{1n}\det(A_{[1n]}) & \text{если} \quad n > 1. \end{cases}$$

Множитель $(-1)^{i+j}\det(A_{[ij]})$ называется алгебраическим дополнением (cofactor) элемента a_{ij}.

Следующие две теоремы (доказательства мы опускаем) указывают основные свойства определителя.

Теорема 31.4 (Свойства определителя)

Определитель квадратной матрицы A обладает следующими свойствами:

- *Если в какой-либо строке или каком-либо столбце матрицы стоят одни нули, то её определитель равен 0.*
- *Если умножить все элементы какой-либо строки матрицы на некоторое число λ, то её определитель умножится на λ.*
- *Если прибавить к элементам одной строки соответствующие элементы другой, то определитель не изменится (аналогично для столбцов).*
- *Определители матриц A и A^T равны.*
- *При перестановке двух строк или столбцов матрицы её определитель меняет знак.*

Если A и B — квадратные матрицы одинакового размера, то $\det(AB) = \det(A)\det(B)$.

Теорема 31.5

Квадратная матрица A вырождена тогда и только тогда, когда $\det(A) = 0$.

Положительно определённые матрицы

Матрица A размера $n\times n$ называется положительно определённой *(positive-definite), если $x^TAx > 0$ для любого ненулевого вектора x размера n. Например, единичная матрица положительно определена, поскольку для вектора $x = (x_1, x_2, \ldots, x_n)^T \neq 0$ мы имеем*

$$\begin{aligned} x^T I_n x &= x^T x \\ &= \|x\|^2 \\ &= \sum_{i=1}^{n} x_i^2 \\ &> 0. \end{aligned}$$

Часто положительно определённые матрицы возникают так:

Теорема 31.6

Для любой матрицы A полного столбцового ранга матрица $A^T A$ положительно определена.

Доказательство.

Покажем, что $x^T(A^T A)x > 0$ для произвольного ненулевого вектора x. В самом деле,

$$\begin{aligned} x^T(A^T A)x &= (Ax)^T(Ax) (\text{Упражнение 31.1-3}) \\ &= \|Ax\|^2 \\ &\geqslant 0 \end{aligned} \qquad (31.8)$$

Выражение $\|Ax\|^2$ представляет собой сумму квадратов элементов вектора Ax. Если $\|Ax\|^2 = 0$, то все элементы вектора Ax равны 0, то есть $Ax = 0$. Но A — матрица полного столбцового ранга, поэтому по теореме 31.2 отсюда следует, что $x = 0$.

Другие свойства положительно определённых матриц указаны разделе 31.6.

Упражнения

31.1-1

Докажите, что произведение двух нижне-треугольных матриц является нижне-треугольной матрицей. Докажите, что определитель нижне- или верхнетреугольной матрицы равен произведению её диагональных элементов. Докажите, что матрица, обратная к нижне-треугольной, сама будут нижне-треугольной (если существует).

31.1-2

Пусть P и A — $(n \times n)$ матрицы, причём P — матрица перестановки. Докажите, что матрица PA получается из A перестановкой строк, а AP — перестановкой столбцов. Докажите, что произведение двух матриц перестановки снова будет матрицей перестановки. Докажите, что если P — матрица перестановки, то P обратима, $P^{-1} = P^T$, и P^T также является матрицей перестановки.

31.1-3

Докажите, что $(AB)^T = B^T A^T$. Докажите, что матрица $A^T A$ является симметрической для любой матрицы A.

31.1-4

Докажите, что обратная матрица единственна: если матрицы B и C являются обратными к матрице A, то $B = C$.

31.1-5

Пусть даны $(n \times n)$-матрицы A и B, для которых $AB = I$. Пусть A' получается из A прибавлением j-й строки к i-й. Докажите, что матрица B', обратная к A', может быть получена вычитанием i-го столбца из j-го в матрице B.

31.1-6

Пусть A — комплексная $(n \times n)$-матрица. Докажите, что все элементы A^{-1} будут вещественными в том и только том случае, если вещественны все элементы A.

31.1-7

Покажите, что если невырожденная матрица A является симметрической, то матрица A^{-1} тоже будет симметрической. Покажите, что для всякой матрицы B надлежащего размера матрица BAB^T будет симметрической.

31.1-8

Покажите, что для матриц полного столбцового ранга, и только для них, из равенства $Ax = 0$ следует равенство $x = 0$. (Указание. Запишите условие линейной зависимости столбцов как матричное уравнение.)

31.1-9

Докажите, что для любых матриц A и B согласованных размеров

$$rank(AB) \leqslant \min(rank(A), rank(B)),$$

причём это неравенство обращается в равенство, если одна из матриц квадратная и невырожденная, (Указание. Воспользуйтесь вторым определением ранга.)

31.1-10

Матрицей Вандермонда *(Vandermonde matrix) называется матрица*

$$V(x_0, x_1, \ldots, x_{n-1}) = \begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \ldots & x_{n-1}^{n-1} \end{pmatrix}.$$

Докажите, что

$$\det(V(x_0, x_1, \ldots, x_{n-1})) = \prod_{0 \leqslant j \leqslant k \leqslant n-1} (x_k - x_j).$$

(Указание. Последовательно полагая $i = n-1, n-2, \ldots, 1$ прибавьте к $(i+1)$-му столбцу i-й, умноженный на $(-x_0)$, а затем примените индукцию.)

31.2 Алгоритм Штрассена умножения матриц

В этом разделе излагается открытый Штрассеном рекурсивный алгоритм, умножающий две $n \times n$ матрицы за время $\Theta(n^{\lg 7}) = O(n^{2.81})$. При достаточно больших n он работает быстрее простейшего алгоритма MATRIX-MULTIPLY *из раздела 26.1.*

Общая схема алгоритма.

Алгоритм Штрассена действует по принципу "разделяй и властвуй". Пусть нужно вычислить произведение двух $(n \times n)$-матриц $C = AB$. Предположив, что n является точной степенью 2, разделим каждую из матриц A,B и C на 4 блока размера $(n/2 \times n/2)$. Перепишем равенство $C = AB$ следующим образом:

$$\begin{pmatrix} r & s \\ l & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix}. \tag{31.9}$$

(В упр. 31.2-2 предлагается рассмотреть случай, в котором n не является точной степенью 2.) Для удобства подматрицы в A идут по алфавиту порядке слева направо, а подматрицы в B — сверху вниз, в соответствии с правилом умножения матриц. Уравнение (31.9) распадается на четыре уравнения

$$r = ae + bf \tag{31.10}$$

$$s = as + bh \tag{31.11}$$

$$t = ce + df \tag{31.12}$$

$$u = cg + dh \tag{31.13}$$

Каждое из четырёх уравнений требует двух умножений $(n/2 \times n/2)$–матриц, после чего произведения складываются. Рассмотрим естественный рекурсивный алгоритм, использующий эти соотношения. Его время работы $T(n)$ (для матриц размера $n \times n$) удовлетворяет неравенству

$$T(n) = 8T(n/2) + \Theta(n^2) \tag{31.14}$$

Отсюда $T(n) = \Theta(n^3)$, но ничего нового мы не получили, так как стандартный алгоритм умножения матриц имеет (асимптотически) то же время работы.

Штрассен придумал, как сэкономить одно умножение и обойтись лишь 7 умножениями $(n/2 \times n/2)$–матриц и $\Theta(n^2)$ операциями сложения и вычитания чисел для умножения $(n \times n)$-матриц. Рекуррентное соотношение тогда принимает вид

$$T(n) = 7T(n/2) + \Theta(n^2), \tag{31.15}$$

откуда

$$\begin{aligned} T(n) &= \Theta(n^{\lg 7}) \\ &= O(n^{2.81}). \end{aligned}$$

Алгоритм Штрассена умножает две $(n \times n)$-матрицы A и B так:

1. Каждая из матриц A и B разбивается на 4 блока, как в (31.9).

2. Строятся 14 матриц $A_1, B_1, A_2, B_2, \ldots, A_7, B_7$. размера $(n/2 \times n/2)$ (для чего нужно $\Theta(n^2)$ операций сложения/вычитания чисел)

3. Рекурсивно вычисляются 7 произведений матриц меньшего размера $P_i = A_iB_i$ $(i = 1, \ldots, 7)$.

4. Вычисляются части r, s, t, u искомой матрицы C. Они являются линейными комбинациями матриц P_i с коэффициентами из множества $\{-1, 0, 1\}$, и вычисление их требует $\Theta(n^2)$ операций сложения/вычитания чисел.

Время работы этого алгоритма, очевидно, удовлетворяет соотношению (31.15). Изложим теперь опущенные детали.

Какие же матрицы меньшего размера нужно перемножать?

Готовые формулы для P_1–P_7 проверить легко, труднее понять, как до них можно догадаться. Вот один из возможных путей.

Будем искать произведения P_i среди выражений вида

$$\begin{aligned} P_i &= A_iB_i \\ &= (\alpha_{i1}a + \alpha_{i2}b + \alpha_{i3}c + \alpha_{i4}d) \cdot (\beta_{i1}e + \beta_{i2}f + \beta_{i3}g + \beta_{i4}h), \end{aligned} \tag{31.16}$$

где коэффициенты α_{ij}, β_{ij} принимают значения $-1, 0, 1$. Конечно, можно искать формулы для P_i и в более общем виде, но оказывается, что этого уже достаточно.

Обратите внимание, что после раскрытия скобок куски матрицы A будут стоять слева в попарных произведениях, а куски матрицы B — справа. Это важно, поскольку умножение матриц не коммутативно.

Для удобства мы будем изображать линейную комбинацию попарных произведений кусков матриц при помощи (4×4)-матрицы, элементами которой служат соответствующие коэффициенты линейной комбинации. Например, равенство (31.10) запишется в виде

формула набрана мной не вполне верно

$$\begin{aligned} r &= ae + bf \\ &= \begin{pmatrix} a & b & c & d \end{pmatrix} \begin{pmatrix} +1 & 0 & 0 & 0 \\ 0 & +1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix} \\ &= \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} e & f & g & h \end{array} \\ \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \end{array}. \end{aligned}$$

Для краткости мы заменяем $+1$ на `+`*, 0 на $\cdot$ и -1 на* `-` *и опускаем имена столбцов и строк. В этих терминах три остальные* *компоненты результирующей матрицы C запишутся так:*

$$\begin{aligned} s &= ag + bh \\ &= \begin{pmatrix} \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}, \\ t &= ce + df \\ &= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \end{pmatrix}, \\ u &= cg + dh \\ &= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}. \end{aligned}$$

[Видно, что ни одну из матриц s, t, u, v нельзя вычислить за одно умножение по формуле (31.16) — нужно два. Если делать это независимо для каждой из четырёх матриц, потребуется 8 умножений. Однако, как мы увидим, одно умножение можно сэкономить, используя одни и те же произведения для нескольких матриц.]

Начнём со следующего наблюдения: s можно вычислить как $s = P_1 + P_2$, где каждая из матриц P_1 и P_2 требует одного умножения:

$$\begin{aligned} P_1 &= A_1 B_1 \\ &= a \cdot (g - h) \\ &= ag - ah \\ &= \begin{pmatrix} \cdot & \cdot & + & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}. \\ P_2 &= A_2 B_2 \\ &= (a + b) \cdot h \\ &= ah + bh \\ &= \begin{pmatrix} \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}. \end{aligned}$$

Аналогичным образом можно вычислить t. Именно, $t = P_3 + P_4$, *где*

$$\begin{aligned} P_3 &= A_3B_3 \\ &= (c+d)\cdot e \\ &= ce + de \\ &= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \end{pmatrix}, \end{aligned}$$

и

$$\begin{aligned} P_4 &= A_4B_4 \\ &= d\cdot(f-e) \\ &= df - de \\ &= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & + & \cdot & \cdot \end{pmatrix}. \end{aligned}$$

Итак, мы израсходовали четыре умножения на вычисление матриц s и t — пока что никакой экономии по сравнению с очевидным порядком действий. Однако по ходу дела мы вычислили произведения, которые нам ещё пригодятся.

Каждое из восьми слагаемых, встречающихся в правой части равенств (31.10)–(31.13) будем называть существенным. *Уже вычисленные две матрицы s и t содержат* 4 *существенных члена ag, bh, ce и df. Остаётся вычислить r и u; они включают в себя* 4 *существенных слагаемых ae, bf, cg и dh, которые соответствуют диагональным позициям* (4×4)*–матрицы), сделав не более трёх умножений. Тем самым одно умножение должно охватывать два существенных слагаемых. Попробуем так:*

$$\begin{aligned} P_5 &= A_5B_5 \\ &= (a+d)\cdot(e+h) \\ &= ae + ah + de + dh \\ &= \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix}. \end{aligned}$$

Помимо двух нужных нам слагаемых ae и dh имеются два лишних: ah и de. Уничтожим их с помощью P_4 и P_2, при этом появятся *два других:*

$$\begin{aligned} P_5 + P_4 - P_2 &= ae + dh + df - bh \\ &= \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & + \end{pmatrix}. \end{aligned}$$

Использовав ещё одно произведение

$$\begin{aligned} P_6 &= A_6 B_6 \\ &= (b - d) \cdot (f + h) \\ &= bf + bh - df - dh \\ &= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & - & \cdot & - \end{pmatrix}, \end{aligned}$$

получим

$$\begin{aligned} r &= P_5 + P_4 - P_2 + P_6 \\ &= ae + bf \\ &= \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}. \end{aligned}$$

Итак, на три матрицы ушло 6 умножений — что же в этом хорошего? А вто что: при симметричном вычислении и мы снова используем P_5 и одно умножение сэкономим. Для сначала сместим лишние слагаемые в P_5 в другом направлении при помощи P_1 и P_3:

$$\begin{aligned} P_5 + P_1 - P_3 &= ae + ag - ce + dh \\ &= \begin{pmatrix} + & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}. \end{aligned}$$

Вычитая дополнительное произведение

$$\begin{aligned} P_7 &= A_7 B_7 \\ &= (a - c) \cdot (e + g) \\ &= ae + ag - ce - cg \\ &= \begin{pmatrix} + & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & - & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}, \end{aligned}$$

получаем

$$\begin{aligned} u &= P_5 + P_1 - P_3 - P_7 \\ &= cg + dh \\ &= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}. \end{aligned}$$

Мы видим, что 7 матриц $P_1, P_2, \ldots, P_7$ позволяют полностью вычислить призведение $C = AB$, что завершает описание алгоритма Штрассена.

Обсуждение

На практике алгоритм Штрассена применяется редко из-за большой величины константы, содержащейся в асимптотическом выражении для времени его работы. Применение его оправдано для плотных (содержащих мало нулей) матриц достаточно большого размера (примерно от 45×45). Небольшие матрицы проще всего перемножать обычным способом, а для больших разреженных (содержащих много нулей) матриц существуют специальные алгоритмы, на практике работающие быстрее штрассеновского. Метод Штрассена, таким образом, представляет интерес в основном с теоретической точки зрения.

Ещё более сложные методы (рассмотрение которых выходит за рамки этой книги) позволяют перемножить две $(n \times n)$–матрицы ещё быстрее. Наилучшая известная оценка составляет приблизительно $O(n^{2.376})$. Что касается нижних оценок, то не известно ничего, кроме тривиальной оценки $\Omega(n^2)$ (по числиу элементов матрицы-результата), так что разрыв между нижними и верхними оценками по-прежнему велик.

Для применения алгоритма Штрассена не обязательно, чтобы элементами матриц были вещественные числа. Важно, чтобы числовая система, которой они принадлежат, являлась кольцом. Однако удаётся использовать идею Штрассена в несколько более общей ситуации; мы рассмотрим этот вопрос в следующем разделе.

Упражнения

31.2-1

Используя алгоритм Штрассена, выполните умножение матриц:

$$\begin{pmatrix} 1 & 3 \\ 5 & 7 \end{pmatrix} \begin{pmatrix} 8 & 4 \\ 6 & 2 \end{pmatrix}.$$

31.2-2

Модифицируйте алгоритм Штрассена, научившись перемножать с его помощью $(n \times n)$–матрицы за время $\Theta(n^{\lg 7})$ при всех значениях n, (а не только для степеней 2).

31.2-3

Представим себе, что мы умеем перемножать две (3×3)*-матрицы, сделав* k *умножений (при этом не используя коммутативности умножения) и используем этот приём рекурсивно для умножения* $(n \times n)$*-матриц, как в методе Штрассена. При каких* k *это позволило бы улучшить оценку Штрассена и умножать* $(n \times n)$*—матрицы за время* $o(n^{\lg 7})$*. Какая оценка при этом получилась бы?*

31.2-4

В.Я. Пан придумал способы умножения двух матриц размера 68×68 *(*132464 *умножений чисел),* 70×70 *(*143640 *умножений) и* 72×72 *(*155424 *умножений). Какой из них даёт лучшую асимптотическую оценку для умножении* $(n \times n)$*-матриц при использовании приёма "разделяй и властвуй"? Сравните эту оценку с оценкой для алгоритма Штрассена.*

31.2-5

Насколько быстро можно умножить $(kn \times n)$*-матрицу на* $(n \times kn)$*-матрицу, используя алгоритм Штрассена в качестве подпрограммы? А сколько времени уйдёт на умножение тех же матриц в обратном порядке?*

31.2-6

Как умножить два комплексных числа $a + bi$ *и* $c + di$*, используя лишь* 3 *операции умножения вещественных чисел? (Алгоритм должен получать на вход значения* a, b, c, d *и вычислять вещественную и мнимую части произведения, то есть* $ac - bd$ *и* $ad + bc$*.)*

31.3 Обращение матриц

На практике решение систем линейных уравнений не требует обращения матриц, как мы сидели в преыдущем разделе (LUP-разложение). Но всё-таки может понадобиться вычислить обратную матрицу, и тогда это можно сделать с помощью того же LUP-разложения. Интерес (скорее теоретический, впрочем) представляет также вопрос о том, как ускорить поиск вычисление обратной матрицы с помоью методов Штрассена. (Именно эту цель преследовал Штрассен в своей работе.)

Вычисление обратной матрицы с помощью LUP-разложения.

Пусть нам дано LUP-разложение $PA = LU$ *матрицы* A *размера* $(n \times n)$*. Зная его, можно решить систему вида* $Ax = b$ *с помощью процедуры* LU-Solve *за время* $\Theta(n^2)$*. Чтобы решить другую систему* $Ax = b'$ *(с той же матрицей, но другой правой частью) нам понадобится ещё столько же времени. Таким образом, зная LUP-разложение матрицы* A*, можно решить* k *систем с матрицей* A *за время* $\Theta(kn^2)$*.*

Матричное уравнение

$$AX = I_n \tag{31.24}$$

для обратной матрицы X можно рассматривать как совокупность n систем вида $Ax = b$. Обозначим i-й столбец матрицы X через X_i; тогда

$$AX_i = e_i, i = 1, \ldots, n$$

поскольку i-м столбцом матрицы I_n является единичный вектор e_i). Другими словами, нахождение обратной матрицы сводится к решению n уравнений с одной матрицей и разными правыми частями. После выполнения LUP-разложения (время $O(n^3)$) на решение каждого из n уравнений нужно время $O(n^2)$, так что и эта часть работы требует времени $O(n^3)$.

Умножение и обращение матриц

Теперь мы покажем, каким образом можно использовать быстрый алгоритм умножения матриц для быстрого вычисления обратной матрицы. (Подчеркнём, что это улучшение имеет скорее теоретический интерес.) Мы докажем, что (при некоторых естественных предположениях) вычисление обратной матрицы имеет ту же сложность, что и умножение матриц того же размера (с точностью до умножения на константу). Сначала мы докажем более простую часть этого утверждения.

Теорема 31.11 (Умножение матриц не сложнее обращения)

Если можно обратить $(n \times n)$-матрицу за время $I(n)$, причём $I(3n) = O(I(n))$, то можно умножить две $(n \times n)$-матрицы за время $O(I(n))$.

Доказательство.

Пусть надо вычислить произведение двух $(n \times n)$-матриц A и B. Составим $(3n \times 3n)$-матрицу

$$D = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}.$$

Матрица, обратная к D, имеет вид

$$D^{-1} = \begin{pmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix},$$

и мы можем вычислить AB как $(n \times n)$-подматрицу в верхнем правом углу матрицы D^{-1}.

Матрицу D можно построить за время $\Theta(n^2) = O(I(n))$ (заметим, что $I(n) \geqslant n^2$, так как нужно вычислить все n^2 элементов обратной матрицы), а обратить за время $O(I(3n)) = O(I(n))$ (согласно условию). Отсюда и следует утверждение теоремы.

Наложенное на $I(n)$ условие $I(3n) = O(I(n))$ означает, что $I(n)$ не делает больших скачков с ростом n. Например, функция $I(n) = \Theta(n^c \lg^d n)$ обладает таким свойством при любых значениях $c > 0, d \geqslant 0$.

Сведение обращения матриц к умножению

Нам понадобятся некоторые свойства симметрических положительно определённых матриц, которые мы докажем в разделе 31.6.

Теорема 31.11 (Обращение матриц не сложнее умножения)

Если можно умножить две $(n \times n)$-матрицы за время $M(n)$, причём $M(n)$ монотонно неубвает и удовлетворяет условию $c_1 M(n) \leqslant M(2n) \leqslant c_2 M(n)$ при некоторых c_1 и c_2, причём $c_1 > 2$, то можно обратить невырожденную $(n \times n)$-матрицу за время $O(M(n))$.

Доказательство. Обращение матрицы сводится к обращению большей матрицы, так как

$$\begin{pmatrix} A & 0 \\ 0 & I_k \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & 0 \\ 0 & I_k \end{pmatrix},$$

для любого $k > 0$. Поэтому мы можем доказывать теорему для n, являющихся степенями двойки, поскольку значения $M(n)$ и $M(n')$, где n' — ближайшая сверху степень двойки, отличаются не более чем в константу раз.

Пусть сначала матрица A, которую нам надо обратить, является положительно определённой симметрической матрицей. Разобьём её на четыре блока размера $n/2 \times n/2$:

$$A = \begin{pmatrix} B & C^T \\ C & D \end{pmatrix}. \tag{31.25}$$

Рассмотрим матрицу

$$S = D - CB^{-1}C^T, \tag{31.26}$$

(дополнение Шура) и напишем соотношение

$$A^{-1} = \begin{pmatrix} B^{-1} + b^{-1}C^T S^{-1} C B^{-1} & -B^{-1} C^T S^{-1} \\ -S^{-1} C B^{-1} & S^{-1} \end{pmatrix} \tag{31.27}$$

(проверяется умножением на A). Поскольку A является положительно определённой симметрической матрицей, то B и S обладают тем же свойством, и потому обратимы (леммы 31.13, 31.14 и 31.15). Легко проверить, что $B^{-1}C^T = (CB^{-1})^T$ и $B^{-1}C^T S^{-1} = (S^{-1}CB^{-1})^T$ (упр. 31.1-3) Соотношения (31.26) и (31.27) задают *рекурсивный алгоритм обращения матрицы, использующий 4 операции умножения $(n/2 \times n/2)$–матриц:*

$$\begin{array}{l} C \cdot B^{-1}, \\ (CB^{-1}) \cdot C^T, \\ S^{-1} \cdot (CB^{-1}), \\ (CB^{-1})^T \cdot (S^{-1}CB^{-1}) \end{array},$$

а также 2 операции обращения матриц того же размера и $O(n^2)$ сложений и других операций. Получаем рекуррентную формулу

$$I(n) \leqslant 2I(n/2) + 4M(n/2) + dn^2$$

для некоторго фиксированного d. Пусть C — достаточно большая константа (насколько, увидим дальше). Тогда можно доказывать неравенство

$$I(n) \leqslant CM(n)$$

по индукции:

$$\begin{aligned} I(n) &\leqslant 2I(n/2) + 4M(n/2) + dn^2 \leqslant \\ &\leqslant 2CM(n/2) + 4M(n/2) + dn^2 \leqslant \\ &\leqslant Cc_1M(n/2) \leqslant \\ &\leqslant CM(n) \end{aligned}$$

Надо только проверить, что при достаточно больших C переход от второй строки к третьей в этом неравенстве законен. В самом деле при больших C основной вклад даёт первое слагаемой второй строки (напомним, что $M(n) \geqslant n^2$, так как нужно вычислить все n^2 элементов произведения), и остаётся вспомнить, что $c_1 > 2$ по условию теоремы.

Итак, с положительно определёнными симметрическими матрицами мы разобрались. Если невырожденная матрицы A не является положительно определённой симметрическрй матрицей, то рассмотрим матрицу A^TA, которая уже будет таковой (упражнению 31.1-3, теорема 31.6). Остаётся заметить, что

$$A^{-1} = (A^TA)^{-1}A^T,$$

поскольку $((A^TA)^{-1}A^T)A = (A^TA)^{-1}(A^TA) = I_n$, а обратная матрица единственна. Видно, что достаточно вычислить произведение A^TA, обратить его (как — мы уже знаем) и умножить результат на матрицу A^T. Каждый из трёх шагов требует времени $O(M(n))$, следовательно, всякую невырожденную матрицу можно обратить за время $O(M(n))$.

Доказательство теоремы 31.12 наводит на мысль о новом способе решения системы $Ax = b$ с невырожденной матрицей A, не *требующем в себя выбора главного элемента. Умножив уравнение слева на (невырожденную) матрицу* A^T, *получим эквивалентное уравнение* $(A^TA)x = A^Tb$. *Матрица* A^TA *является положительно определённой симметрической матрицей, и для неё можно найти* LU*-разложение с помощью процедуры* LU-Decomposition, *а затем решить систему с правой частью* A^Tb, *используя прямую и обратную подстановку. Всё это, конечно, так, но на практике лучше применить процедуру* LUP-Decomposition *к исходной матрице* A *при этом константа в оценке для числа операций меньше, и ошибки округления меньше сказываются на результате.*

Упражнения

31.5-1

Докажите, что умножение матриц столь же трудно, как и возведение матрицы в квадрат: докажите что если $M(n)$ *— время, требуемое для умножения двух* $(n \times n)N$*матриц, а* $S(n)$ *— время, нужное для возведения* $(n \times n)$*-матрицы в квадрат, то* $S(n) = \Theta(M(n))$.

31.5-2

Докажите, что умножение матриц столь же трудно, как и вычисление LUP*-разложения. Формально говоря, докажите что если* $M(n)$ *— время, требуемое для умножения двух* $(n \times n)$*-матриц, а* $L(n)$ *— время, нужное для вычисления* LUP*-разложения* $(n \times n)$*-матрицы, то* $L(n) = \Theta(M(n))$.

31.5-3

Докажите, что вычисление определителя не труднее умножения матриц: докажите что если $M(n)$ *— время, требуемое для умножения двух* $(n\times n)$*-матриц, то определитель* $(n\times n)$*-матрицы можно вычислить за время* $D(n) = O(M(n))$.

31.5-4

Пусть $M(n))$ *— время, нужное для умножения двух булевых матриц размера* $n\times n$, *а* $T(n)$ *— время, нужное для вычисления транзитивного замыкания булевой* $(n \times n)-$*матрицы. Покажите, что* $M(n) = O(T(n))$ *и* $T(n) = O(M(n)\lg n)$ *при некоторых естественных предположениях на* M *и* T.

31.5-5

Применим ли метод обращения матриц из доказательства теоремы 31.12 к матрицам над полем вычетов по модулю 2? Почему?

*31.5-6**

Как обобщить алгоритм из доказательства теоремы 31.12 на случай матриц над полем комплексных чисел?

(Указание. Транспонирование матрицы заменяется сопряжением*: матрица* A^* *(conjugate transpose) получается комплексным сопряжением всех элементов в* A^T. *Роль симметрических матрицы играют* эрмитовы, *для которых* $A = A^*$.*)*

31.4 Положительно определённые симметрические матрицы и метод наименьших квадратов

Это очень важный класс матриц, и они обладают разными полежными свойствами. Например, такая матрица не может быть вырожденной; при построении LU-разложения можно не проводить выбора главного элемента — всё равно деления на ноль не будет, как мы увидим. В этом разделе мы покажем, как их можно использовать для так называемого приближения методом наименьших квадратов. Начнём с такого важного свойства:

Лемма 31.13

Любая положительно определённая симметрическая матрица является невырожденной.

Доказательство.

Пусть матрица A вырождена и x — ненулевой вектор, для которого $Ax = 0$ (следствие 31.3). Тогда $x^T Ax = 0$, и потому матрица A не может быть положительно определённой.

Теперь перейдем к более сложному вопросу: почему процедура LU-Decomposition *для случая положительно определённых симметрических матриц обходится без деления на 0. Нам понадобится понятие k-го* углового минора *(leading submatrix) матрицы A. Он определяется как подматрица, стоящая на пересечении первых k строк и первых k столбцов матрицы A; обозначим его A_k.*

Лемма 31.14

Всякий угловой минор положительно определённой симметрической матрицы сам являетя положительно определённой симметрической матрицей.

Доказательство.

Симметричность очевидна. Для доказательства положительной определённости минора A_k возьмём произвольный k-элементный вектор x. Разбив матрицу матрицы A на блоки

$$A = \begin{pmatrix} A_k & B^T \\ B & C \end{pmatrix}.$$

и применив условие положительной определённости к вектору, дополненному нулями, получаем требуемое:

$$\begin{aligned} x^T A_k x &= \begin{pmatrix} x^T & 0 \end{pmatrix} \begin{pmatrix} A_k & B^T \\ B & C \end{pmatrix} \begin{pmatrix} x \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} x^T & 0 \end{pmatrix} A \begin{pmatrix} x \\ 0 \end{pmatrix} \\ &> 0. \end{aligned}$$

Теперь посмотрим на дополнение Шура для положительно определённых матриц. Пусть A_k — главный минор положительно определённой симметрической матрицы A. Разобьём A на блоки следующим образом:

$$A = \begin{pmatrix} A_k & B^T \\ B & C \end{pmatrix}. \qquad (31/28)$$

Дополнением Шура *к подматрице A_k матрицы A (Schur complement of A with respect to A_k) будем называть матрицу*

$$S = C - BA_k^{-1}B^T. \qquad (31.29)$$

По лемме 31.14 матрица A_k является положительно определённой симметрической матрицей, поэтому (лемма 31.13) A_k^{-1} существует. Обратите внимание, что при $k = 1$ это определение согласуется с прежним определением (31.23).

Докажем результат, использованный при доказательстве теоремы 31.12.

Лемма 31.15 (о дополнении Шура)

Если A является положительно определённой симметрической матрицей, а A_k — её k–й угловой минор, то дополнение Шура к подматрице A_k матрицы A само будет симметрической положительно определённой матрицей.

Доказательство.

Как следует из упражнения 31.1-7, матрица S будет симметрической; остаётся показать её положительную определённость. Мы знаем, что $x^TAx > 0$ для любого ненулевого вектора x. Разбив вектор x на части y и z (в соответствии с разбиением (31.28) матрицы A), мы можем написать следующее тождество (матрица A_k обратима, как мы знаем):

$$\begin{aligned} x^TAx &= \begin{pmatrix} y^T & z^T \end{pmatrix} \begin{pmatrix} A_k & B^T \\ B & C \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix} \\ &= y^TA_ky + y^TB^Tz + z^TBy + Z^TCz \\ &= (y + A_k^{-1}B^Tz)^TA_k(y + A_k^{-1}B^Tz) + z^T(C - BA_k^{-1}B^T)z, \end{aligned} \qquad (31.30)$$

(Последнее равенство соответствует выделению полного квадрата, как мы увидим в упр. 31.6-2.)

Теперь для любого z можно подобрать такое y, чтобы первое слагаемое в соотношении (31.30) обратилось в 0 — а значит, остающееся второе слагаемое

$$z^T(C - BA_k^{-1}B^T)z = z^TSz$$

положительно для любого z. Положительная определённость S доказана.

Следствие 31.16

Для положительно определённой симметрической матрицы процедура LU-Decomposition *не сталкивается с делением на* 0.

Доказательство.

Пусть A — положительно определённая симметрическая матрица. Докажем, что все главные элементы по ходу действия будут положительны (и тем самым не равны 0*). Первый из них (a_{11}) положителен, поскольку по определению положительно определённой матрицы $a_{11} = e_1^T A e_1^T > 0$. На следующем шаге рекурсии алгоритм применяется к дополнению Шура, которое по лемме 31.15 само является положительно определённой симметрической матрицей, так что и все дальнейшие главные элементы положительны (индукция).*

Метод наименьших квадратов.

Метод наименьших квадратов использует положительно определённые симметрические матрицы для отыскания кривых данного вида, походящих поблизости от заданных точек. Пусть нам задан набор точек

$$(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$$

($x_i \neq x_j$ при $i \neq j$), причём значения y_i считаются содержащими ошибки измерения. Мы ищем функцию $F(x)$, для которой

$$y_i = F(x_i) + \eta_i, \tag{31.31}$$

при $i = 1, \dots, m$, причём погрешности η_i малы. Ищем мы её среди функций определенного класса, а именно, среди линейных комбинаций

$$\sum_{j=1}^{n} c_j f_j(x).$$

заранее выбранных базисных функций *(basic functions) f_j. Часто в качестве базисных функций рассматривают мономы $f_j(x) = x^{j-1}$ вплоть до некоторой фиксированной степени; другими словами, мы ищем F среди многочленов степени не выше $n - 1$:*

$$c_1 + c_2 x + c_3 x^2 + \dots + c_n x^{n-1}.$$

При $n = m$ можно найти многочлен, в точности проходящий через заданные точки, то есть удовлетворяющий соотношению (31.31) с нулевыми погрешностями. Как правило, такой подход оказывается неудачным, так как ошибки измерений сильно влияют на значения многочлена вне $x_1, x_2, \dots, x_n$, и получается ерунда. Разумнее выбрать n много меньшим m — тогда есть шанс, что "шум", вносимый ошибками измерения, отфильтруется. Мы не обсуждать практические правила выбора n, а будет считать, что n уже задано. Мы получаем систему, где уравнений больше, чем неизвестных, зато их не надо решать точно —

если левая часть близка к правой, уже хорошо. Как поступать с такой системой?

Пусть A — матрица значений базисных функций в заданных точках:

$$A = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \dots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \dots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \dots & f_n(x_m) \end{pmatrix},$$

а c — вектор из n искомых коэффициентов: $c = (c_k)$. Тогда

$$Ac = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \dots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \dots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \dots & f_n(x_m) \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}$$
$$= \begin{pmatrix} F(x_1) \\ F(x_2) \\ \vdots \\ F(x_m) \end{pmatrix}$$

будет вектором из m значений, через которые проходит кривая.

Мы хотим, чтобы вектор невязки *(approximation error)*

$$\eta = Ac - y,$$

(размера m) был как можно меньше. Здесь "меньше" мы понимаем как "короче", вычисляя длину по формуле

$$\|\eta\| = \left(\sum_{i=1}^{m} \eta_i^2\right)^{1/2}$$

(евклидова норма). Другими словами, мы хотим, чтобы сумма квадратов невязок была минимальной, отсюда и название метод наименьших квадратов *(least squares). Как учат в курсе анализа, для поиска минимума надо продифференцировать*

$$\|\eta\|^2 = \|Ac - y\|^2 = \sum_{i=1}^{n} \left(\sum_{j=1}^{n} a_{ij}c_j - y_i\right)^2$$

по всем переменным c_k

$$\frac{\partial\|\eta\|^2}{\partial c_k} = \sum_{i=1}^{n} 2\left(\sum_{j=1}^{n} a_{ij}c_j - y_i\right) a_{ik} = 0. \qquad (31.32)$$

Эту систему из n уравнений ($k = 1, 2, \ldots, n$) можно записать как матричноме уравнение

$$(Ac - y)^T A = 0,$$

которое (см. упр. 31.1-3) эквивалентно уравнению

$$A^T (Ac - y) = 0.$$

Раскрыв скобки, мы приходим к уравнению

$$A^T Ac = A^T y, \tag{31.33}$$

называемому в математической статистике нормальным уравнением (normal equation). Матрица $A^T A$ будет симметрической (упр. 31.1-3), и, если только A имеет полный столбцовый ранг, положительно определённой (теорема 31.6). В этом случае (лемма 31.13) существует обратная матрица $(A^T A)^{-1}$, и решение системы (31.33) единственно:

$$\begin{aligned} c &= ((A^T A)^{-1} A^T) y \\ &= A^+ y, \end{aligned} \tag{31.34}$$

Здесь через A^+ обозначена матрица $(A^T A)^{-1} A^T$, называемая псевдообратной (pseudoinverse) к матрице A. Это понятие является естественным обобщением понятия обратной матрицы на случай неквадратных матрицы. (Сравните формулу (31.34), дающую решение системы $Ac = y$ по методу наименьших квадратов, с формулой $x = A^{-1} b$, дающей точное решение системы $Ax = b$.)

Для примера рассмотрим 5 экспериментальных точек

$$(-1, 2), (1, 1), (2, 1), (3, 0), (5, 3)$$

(чёрные кружки рис. 31.3.) Мы хотим провести рядом с ними график квадратного трёхчлена

$$F(x) = c_1 + c_2 x + c_3 x^2.$$

Выпишем матрицу значений базисных функций:

$$A = \begin{pmatrix} 1 & x_1 & x_1^2 \\ 2 & x_2 & x_2^2 \\ 3 & x_3 & x_3^2 \\ 4 & x_4 & x_4^2 \\ 5 & x_5 & x_5^2 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 5 & 25 \end{pmatrix}$$

и найдём псевдообратную к ней:

$$A^+ = \begin{pmatrix} 0.500 & 0.300 & 0.200 & 0.100 & -0.100 \\ -0.388 & 0.093 & 0.190 & 0.193 & -0.088 \\ 0.060 & -0.036 & -0.048 & -0.036 & 0.060 \end{pmatrix}.$$

Рисунок 31.1 31.3 Метод наименьших квадратов в применении к точкам $\{(-1,2),(1,1),(2,1),(3,0),(5,3)\}$ (чёрные). Показан наилучший квадратный трёхчлен и его значения (светлые кружки). Серые отрезки — невязки, их сумма квадратов должна быть как можно меньше.

Искомый вектор коэффициентов ($c = A^{+}y$) будет равен

$$c = \begin{pmatrix} 1.200 \\ -0.757 \\ 0.214 \end{pmatrix}.$$

Ответ: квадратный трёхчлен

$$F(x) = 1.200 - 0.757x + 0.214x^2,$$

представляет собой наилучшее приближение в смысле наименьших квадратов.

На практике при решении нормального уравнения (31.33) строят LU-разложение матрицы $A^T A$ — это положительно определённая симметрическая матрица (если A имеет полный ранг, см. упр. 31.1-3 и теорему 31.6).

Упражнения

31.6-1

Докажите, что все элементы на диагонали положительно определённой симметрической матрицы положительны.

31.6-2

Докажите, что для положительно определённой симметрической матрицы размера 2×2 $\begin{pmatrix} a & b \\ b & c \end{pmatrix}$ дискриминант $ac-b^2$ положителен, выделив полный квадрат в соответствующей квадратичной форме (аналогично доказательству Леммы 31.15). Как вывести это из утверждения леммы 31.15?

31.6-3

Докажите, что наибольший элемент положительно определённой матрицы матрицы находится на её диагонали.

31.6-4

Докажите, что определитель всякого углового минора положительно определённой симметрической матрицы положителен.

31.6-5

Пусть A_k — k-й угловой минор положительно определённой симметрической матрицы A. Докажите, что в при построении LU-разложения матрицы A с помощью LU-Decomposition *k-й главный элемент будет равен $\det(A_k)/\det(A_{k-1})$. (Мы полагаем условно $\det(A_0) = 1$.)*

31.6-6

Даны точки $(1,1),(2,1),(3,3),(4,8)$. Постройте приближение вида

$$F(x) = c_1 + c_2 x \lg x + c_3 e^x$$

методом наименьших квадратов.

31.6-7

Докажите следующие свойства псевдообратных матриц:

$$\begin{aligned} AA^+A &= A, \\ A^+AA^+ &= A^+, \\ (AA^+)^T &= AA^+, \\ (A^+A)^T &= A^+A. \end{aligned}$$

Задачи

31-1 Вероятностный алгоритм Шамира для умножения булевых матриц.

В разделе 31.3 мы видели, что алгоритм Штрассена нельзя просто так применить для булевых матриц, поскольку булево квазикольцо $Q = (\{0,1\}, \vee, \wedge, 0, 1)$ *не является кольцом. Теорема 31.10 модифицирует алгоритм Штрассена и позволяет умножить две булевые матрицы размера* $n \times n$ *за время* $O(n^{\lg 7})$*, но арифметические операции приходится выполнять над* $O(\lg n)$*-битовыми числами. Вероятностный алгоритм Шамира производит только битовые операции и работает почти столь же быстро, однако не всегда даёт правильный ответ. Сейчас мы его опишем.*

a. *Покажите, что числовая система* $R = (\{0,1\}, \oplus, \wedge, 0, 1)$*, где* $\oplus$ *означает XOR (исключаещее ИЛИ, сложение по модулю 2) представляет собой кольцо.*

Пусть A *и* B *— булевы* $(n \times n)$*−матрицы, а* $C = AB$ *— их произведение над квазикольцом* Q *Построим по матрице* A *матрицу* A' *так: нули остаются на своих местах, а каждая единица либо остаётся на месте, либо заменяется нулём (с вероятностью 50%, разные единицы ведут себя независимо).*

b. *Положим* $C' = (c'_{ij}) = A'B$*, причём умножение выполняется над кольцом* R*. Докажите, что если* $_{ij} = 0$*, то* $'_{ij} = 0$*. Докажите, что если* $c_{ij} = 1$*, то* $'_{ij} = 1$ *с вероятностью* $1/2$*.*

c. *Взяв произвольное* $\varepsilon > 0$*, повторим процедуру построения* A' *и вычисления* C' *чем* $\lg(n^2/\varepsilon)$ *раз или более, каждый раз делая независимые случайные выборы. Докажите, что если* $c_{ij} = 1$ *для некоторых* i *и* j*, то вероятность того, что* c'_{ij} *ни разу не примет значения 1, не превосходит* ε/n^2*. Докажите, что вероятность того, что в любой позиции, где* $c_{ij} = 1$*, хоть раз был правильный ответ (т.е. 1), не меньшее* $1 - \varepsilon$*.*

d. *Для любой константы* k *укажите алгоритм, вычисляющий произведение двух булевых* $(n \times n)$*-матриц за время* $O(n^{\lg 7} \lg n)$ *с вероятностью ошибки не выше* $1/n^k$*. Над элементами матриц разрешается производить только побитовые операции* $\wedge$*,* $\vee$ *и* $\oplus$*.*

31-2 Трёхдиагональные системы линейных уравнений.

Рассмотрим трёхдиагональную матрицу

$$A = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix}.$$

a. *Найдите LU-разложение матрицы A.*

b. *Используя прямую и обратную подстановку, решите систему $Ax = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \end{pmatrix}^T$.*

c. *Найдите обратную матрицу A^{-1}.*

d. *Покажите, как решать систему $Ax = b$ с положительно определённой симметрической трёхдиагональной $(n \times n)$-матрицей A за время $O(n)$, используя LU-разложение. Объясните, почему любой алгоритм, основанный на обращении матрицы A, имеет асимптотически худшую оценку сложности.*

e. *Покажите, как решать систему $Ax = b$ с невырожденной трёхдиагональной $(n \times n)$-матрицей A за время $O(n)$, используя LUP-разложение.*

31-3 Сплайны

Проводя кривые через заданные точки, часто используют кубические сплайны *(cubic splines). Пусть задан набор $(n+1)$ точек $\{(x_i, y_i) : i = 0, 1, \ldots, n\}$, причём $x_0 < x_1 < \cdots < x_n$. Мы хотим провести через все эти точки кривую, состоящую из кусков n кубических полиномов, на каждом отрезке свой. Когда x пробегает отрезок $[x_i, x_{i+1}]$ ($i = 0, 1, \ldots, n$), сплайн f определяется равенством $f(x) = f_i(x - x_i)$, где $f_i(x) = a_i + b_i x + c_i x^2 + d_i x^3$ — кубический полином. Точки x_i, в которых куски состыковываются, называются* узлами *(knots) Для простоты будем предполагать, что $x_i = i$ при $i = 0, 1, \ldots, n$.*

Потребовав от f непрерывности, получим условия:

$$f(x_i) = f_i(0) = y_i,$$
$$f(x_{i+1}) = f_i(1) = y_{i+1},$$

для $i = 0, 1, \ldots, n-1$. Отсутствие изломов (разрывов первой производной) в узлах даёт условия

$$f'(x_{i+1}) = f_i'(1) = f_{i+1}'(0),$$

для $i = 0, 1, \ldots, n-1$.

a. *Пусть мы каким-то образом выбрали, помимо самих значений $y_i = f(x_i)$, ещё и производные $D_i = f'(x_i)$ в каждом узле. Выразите коэффициенты a_i, b_i, c_i и d_i через значения y_i, y_{i+1}, D_i и D_{i+1}. (Напоминаем, что $x_i = i$.) Сколько времени понадобится для такого вычисления?*

Если значения производной в узлах не заданы, встаёт вопрос об их выборе. Один из методов состоит в том, чтобы потребовать непрерывности вплость до второй производной f''. Это даёт набор условий:

$$f''(x_{i+1}) = f_i''(1) = f_{i+1}'(0),$$

для $i = 0, 1, \ldots, n-1$. Пока что значения второй производной f в точках x_0 и x_n никак не ограничиваются; положив $f''(x_0) = f_0''(0) = 0$ и $f''(x_n) = f_n''(1) = 0$, получаем так называемый естественный *кубический сплайн (natural cubic spline).*

b. *Используя непрерывность вплоть до второй производной, покажите, что*

$$D_{i-1} + 4D_i + D_{i+1} = 3(y_{i+1} - y_{i-1}) \tag{31.35}$$

для $i = 1, \ldots, n-1$.

c. *Для естественных кубических сплайнов докажите равенства*

$$2D_0 + 4D_1 = 3(y_1 - y_0), \tag{31.36}$$

$$D_{n-1} + 2D_n = 3(y_n - y_{n-1}). \tag{31.37}$$

d. *Перепишите уравнения (31.35)-(31.37) в виде матричного уравнения на вектор неизвестных $D = (D_0, D_1, \ldots, D_n)$. Какими свойствами обладает матрица этого уравнения?*

e. *Докажите, что естественный кубический сплайн для $n+1$ точек можно построить за время $O(n)$ (ср. задачу 31-2).*

f. *Как построить естественный кубический сплайн для $n+1$ точек $(x_0, y_0), \ldots (x_n, y_n)$, у которых $x_0 < x_1 < \cdots < x_n$, (но не обязательно $x_i = i$). Какое матричное уравнение придётся решать и сколько времени для этого потребуется?*

Замечания

Теория численных методов — целая наука, которой мы едва коснулись. Мы особенно рекомендуем следующие учебники: Джордж и Лью [18], Голуб и ван Лоан [89], Пресс, Флэнери, Тьюкольски и Веттерлинг [161,162], Стренг [181,182].

Появление работы Штрассена [183] в 1969 году было большим сюрпризом — никто не ожидал, что обычный алгоритм умножения матриц не оптимален. С тех пор асимптотическая верхняя оценка сложности умножения матриц многократно улучшалась и достигла $O(n^{2.376})$ (Копперсмит и Виноград [52]). Наглядное представление алгоритма Штрассена картинками с плюсами и минусами было использовано в работе Патерсона [15]. Фишер и Мейер [67] применили алгоритм Штрассена для умножения булевых матриц (теорема 31.10 этого раздела)

Метод Гаусса исключения неизвестных, применяющийся при построении LU- и LUP-разложения, был первым систематическим методом решения систем линейных уравнений (и одним из *первых численных алгоритмов). Его изобретение обычно связывают с именем К.Ф. Гаусса (1777-1855), хотя он был известен и раньше.*

Алгоритм обращения за время время $O(n^{\lg 7})$ *содержится в классической работе Штрассена [183]. Виноград [203] заметил, что В своей уже упоминавшейся работе [183] что умножение матриц не сложнее обращения; обратную оценку получили Ахо, Хопкрофт и Ульман [4].*

Прекрасное изложение теории положительно определённых симметрических матриц (и вообще линейной алгебры) имеется в книге Стренга [182]. На с. 334 её автор пишет: "Меня часто спрашивают, бывают ли несимметричные положительно определённых матрицах. Я никогда не использую этого термина."

32. Многочлены и быстрое преобразование Фурье

Стандартные способы сложения и умножения двух многочленов степени n требуют времени $\Theta(n)$ и $\Theta(n^2)$ соответственно. В этой главе описывается, как уменьшить время умножения до $\Theta(n \lg n)$ при помощи быстрого преобразования Фурье (Fast Fourier Transform, FFT).

Многочлены

Многочлен *(polynomial) $A(x)$ от переменной x над полем F (polynomial in the variable x over F) имеет вид*

$$A(x) = \sum_{j=0}^{n-1} a_j x^j.$$

Значения a_0, a_1, ... , a_{n-1} принадлежат полю F (в большинстве наших примеров F будет полем комплексных чисел $\mathbb{C}$). Они называются коэффициентами *(coefficients) многочлена. Наибольший из показателей степеней с ненулевыми коэффициентами называется* степенью *(degree) многочлена. Таким образом, наша формула дает общий вид многочлена степени меньше n (polynomial of degree-bound n).*

Пусть $A(x)$ и $B(x)$ — два многочлена степени меньше n. Если мы их сложим, то получим их сумму *(sum), многочлен $C(x)$ степени меньше n, коэффициенты которого есть суммы соответствующих коэффициентов $A(x)$ и $B(x)$. В этом случае $C(x) = A(x) + B(x)$ для всех x из поля F. Таким образом, если*

$$A(x) = \sum_{j=0}^{n-1} a_j x^j,$$

а

$$B(x) = \sum_{j=0}^{n-1} b_j x^j,$$

то

$$(x) = \sum_{j=0}^{n-1} c_j x^j,$$

где $c_j = a_j + b_j$ для $j = 0, 1, \ldots, n-1$. Например, если $A(x) = 6x^3 + 7x^2 - 10x + 9$, а $B(x) = -2x^3 + 4x - 5$, тогда $C(x) = 4x^3 + 7x^2 - 6x + 4$.

Результатом умножения двух многочленов $A(x)$ и $B(x)$ будет их произведение *(product) — многочлен $C(x)$, для которого $C(x) = A(x)B(x)$ для всех x из поля F. Умножение выполняется так: каждое слагаемое многочлена $A(x)$ умножается на каждое слагаемое многочлена $B(x)$, после чего выполняется приведение подобных членов (слагаемых с одинаковыми степенями). Например, мы* можем умножить $A(x) = 6x^3+7x^2-10x+9$ на $B(x) = -2x^3+4x-5$ следующим образом:

[здесь приводится умножение столбиком - надо взять его из оригинальных файлов]

```
%                        6x^3 +  7x^2 - 10x +  9
%                       -2x^3         +  4x -  5
%                       ------------------------
%                      -30x^3 - 35x^2 + 50x - 45
%                 24x^4 +28x^3 - 40x^2 + 36x
%  -12x^6 -14x^5 +20x^4 -18x^3
%  ----------------------------------------------
%  -12x^6 -14x^5 +44x^4 -20x^3 - 75x^2 + 86x - 45
```

Другими словами, произведение $C(x)$ двух многочленов $A(x)$ и $B(x)$ степени меньше n определяется как

$$C(x) = \sum_{j=0}^{2n-2} c_j x^j, \qquad (32.1)$$

где

$$c_j = \sum_{k=0}^{j} a_k b_{j-k}. \qquad (32.2)$$

Степень многочлена-произведения равна сумма степеней сомножителей:

$$degree(C) = degree(A) + degree(B).$$

Поэтому произведение многочленов степеней меньше m и n будет многочленом степени меньше $m+n-1$ (и тем более меньше $m+n$).

План главы

В разделе 32.1 рассматриваются две способа представления многочленов: как набор коэффициентов и с помощью набора значений в заданных точках. Выполнение умножения по формуле (32.2) требует времени $\Theta(n^2)$, если многочлены задаются своими коэффициентами. Если многочлены задаются своими значениями в некоторых точках, то для умножения достаточно времени $O(n)$ (умножаем значения поточечно). Используя переход от одного представления к другому и поточечное умножение, мы сможем выполнять умножение (находить коэффициенты произведения по заданным коэффициентам сомножителей) за время $\Theta(n \lg n)$. Для этого нам потребуются комплексные корни из единицы, преобразование Фурье и обратное к нему (раздел 32.2). В разделе 32.3 описываются быстрые последовательные и параллельные реализации преобразования Фурье.

Поскольку мы постоянно используем комплексные числа, в этом разделе символ i зарезервирован для мнимой единицы ($\sqrt{-1}$).

32.1 Представление многочленов

Многочлен можно задать, указав его коэффициенты, а можно — набором его значений в некотором числе точек. Переход от одного способа к другому будет использован для умножения двух многочленов степени меньше n за время $\Theta(n \lg n)$.

Задание многочлена вектором коэффициентов

Многочлен $A(x) = \sum\limits_{j=0}^{n-1} a_j x^j$ *может быть задан вектором коэффициентов* $a = (a_0, a_1, \ldots, a_{n-1})$ *(coefficient representation). (В матричных уравнениях этого раздела мы будем использовать в основном векторы-столбцы.)*

Представление многочлена коэффициентами удобно в целом ряде случаев. Например, операция вычисления значения *многочлена $A(x)$ в данной точке x_0 (evaluating the polynomial at a given point) может быть выполнена за время $\Theta(n)$ по* схеме Горнера *(Horner's rule):*

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \cdots + x_0(a_{n-1}))\ldots)).$$

Сложение многочленов, заданных векторами коэффициентов, также требует времени $\Theta(n)$: если многочлены заданы векторами коэффициентов $a = (a_0, a_1, \ldots, a_{n-1})$ и $b = (b_0, b_1, \ldots, b_{n-1})$, то их сумма задается вектором $c = (c_0, c_1, \ldots, c_{n-1})$, в котором $c_j = a_j + b_j$ для $j = 0, 1, \ldots, n-1$.

Теперь рассмотрим умножение двух многочленов $A(x)$ и $B(x)$, степени ниже n. Если непосредственно использовать формулу 32.2), то на вычисления потребуется время $\Theta(n^2)$, поскольку каждый коэффициент вектора a надо умножить на каждый коэффициент вектора b. Поэтому важно научиться экономить при умножении — самой трудоемкой из рассмотренных операций. Другими словами, мы хотим быстро вычислять вектор c, заданный уравнением (32.2). Он называется свёрткой *(convolution) векторов a и b и обозначается $c = a \otimes b$. Вычисление свёртки (иными словами, умножение многочленов, заданных векторами коэффициентов) часто встречается на практике.*

Задание многочлена набором значений

Фиксируем n различных точек $x_0, x_1, \ldots, x_{n-1}$. Многочлен $A(x)$ степени ниже n однозначно определяется своими значениями в этих точках, то есть набором из n пар аргумент-значение

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\},$$

где

$$y_k = A(x_k) \tag{32.3}$$

для $k = 0, 1, \ldots, n-1$. Таким образом, для каждого набора точек $x_0, x_1, \ldots, x_{n-1}$ мы получаем свой способ представления многочленов с помощью значений в этих точках.

Переход от коэффициентов многочлена $A(x)$ к его значениям требует времени $\Theta(n^2)$, если мы вычисляем отдельно значение в каждой из n точек по схеме Горнера за $\Theta(n)$ шагов. В дальнейшем мы увидим, что время можно сократить — при подходящем выборе точек достаточно $O(n \lg n)$ операций.

Обратный переход — от набора значений многочлена к его коэффициентам — называется интерполяцией *(interpolation). Следующая теорема утверждает, что интерполяция выполняется однозначно, если степень многочлена меньше числа точек интерполяции.*

Теорема 32.1 (Однозначность интерполяции)

Для любого множества $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$ пар аргумент-значение (все x_i различны) существует единственный многочлен $A(x)$ степени ниже n, для которого $y_k = A(x_k)$ для $k = 0, 1, \ldots, n-1$.

Доказательство

Теорема утверждает, что некоторая матрица обратима. В самом деле, уравнение (32.3) можно записать в матричном виде:

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \ldots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} \qquad (32.4)$$

Левая матрица называется матрицей Вандермонда (Vandermonde matrix) и обозначается $V(x_0, x_1, \ldots, x_{n-1})$. Как утверждает упражнение 31.1-10, определитель этой матрицы равен

$$\prod_{j<k} (x_k - x_j),$$

и по теореме 31.5 эта матрица является обратимой (невырожденной), если x_k различны. Поэтому коэффициенты многочлена a_j однозначно определяются по формуле

$$a = V(x_0, x_1, \ldots, x_{n-1})^{-1} y.$$

Это доказательство сводит задачу интерполяции к решению системы линейных уравнений (32.4). Если делать это по общим правилам решения систем линейных уравнений, описанным в главе 31 (LU-разложение), потребуется время $O(n^3)$. Более быстрый алгоритм интерполяции основывается на формуле Лагранжа:

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)}. \qquad (32.5)$$

Убедитесь, что правая часть (32.5) представляет собой многочлен степени меньше n *и что* $A(x_k) = y_k$ *для всех* k. *В упражнении 32.1-4 предлагается показать, что с помощью формулы Лагранжа можно вычислить коэффициенты многочлена* A *за время* $\Theta(n^2)$.

Таким образом, мы можем переходить от набора n *коэффициентов к набору значений в* n *точках и обратно за* $O(n^2)$ *операций. (Интерполяция является вычислительно неустойчивой операцией. Хотя описанный здесь подход математически корректен, надо понимать, что небольшое изменение входных значений или ошибки округления во время вычислений могут вызвать сильное изменение результата.)*

Представление многочлена с помощью значений в заданных точках удобно для многих операций: например, для сложения достаточно сложить значения многочленов в каждой из точек. Другими словами, если многочлен $A(x)$ *задан набором пар*

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\},$$

а многочлен B *— набором*

$$\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{n-1}, y'_{n-1})\},$$

(заметьте, что значения A *и* B *заданы в одних и тех же* n *точках), то многочлен* $C(x)$ *задается парами*

$$\{(x_0, y_0 + y'_0), (x_1, y_1 + y'_1), \ldots, (x_{n-1}, y_{n-1} + y'_{n-1})\}.$$

Подобным образом можно поступать и с умножением, перемножая значения в каждой точке по отдельности. Надо иметь в виду, однако, что при умножении степени многочленов складываются, и произведение двух многочленов степени меньше n *может иметь степень больше* n. *Она заведомо меньше* $2n$ *(даже* $2n - 1$*), так что для восстановления произведения достаточно* $2n$ *точек (теорема 32.1). Таким образом, умножая два многочлена* A *и* B *степени меньше* n, *полезно с самого начала иметь значения многочленов* A *и* B *не в* n, *а в* $2n$ *точках (одних и тех для для* A *и* B*). После этого эти значения можно перемножить за за время* $\Theta(n)$ *и получить представление произведения* $C = AB$ *в виде набора пар аргумент–значение. (Сравните это время с* $\Theta(n^2)$ *при вычислении коэффициентов произведения по формуле (32.2).)*

Осталось обсудить такой вопрос: мы знаем значения многочлена в заданных точках; как найти его значение в точке, которой нет среди них? По-видимому, нет более простого способа сделать это, чем сначала найти коэффициенты многочлена, а затем вычислить его значение в нужной точке.

Быстрое умножение многочленов, заданных вектором коэффициентов

Надписи на самой картинке
на стрелках:
Обычное умножение, время $\Theta(n^2)$
Вычисление значений, время $\Theta(n \lg n)$
Интерполяция, время $\Theta(n \lg n)$
Поточечное умножение, время $\Theta(n)$
в правом столбце:
Представление набором коэффициентов
Представление набором значений

Рисунок 31.2 32.1 Схема быстрого алгоритма умножения многочленов. В верхней части рисунка многочлены заданы векторами коэффициентов, в нижней — значениями. Стрелки, идущие слева направо, соответствуют умножению. Символы ω_{2n}^{i} обозначают комплексные корни из единицы степени $2n$.

Если научиться быстро переходить от коэффициентам к значениям и обратно, то появится возможность использовать возможность за линейное время умножить значения в данных точках для быстрого умножения многочленов, заданных вектором коэффициентов (от коэффициентов переходим к значениям — перемножаем — переходим обратно).

При этом можно использовать любой набор из n различных точек, но выбрав их удобным образом, можно сократить время преобразования в ту и другую сторону до $\Theta(n \lg n)$*. Как мы увидим в разделе 32.2, удобно взять в качестве точек комплексные корни из единицы, в этом случае оба перехода сведутся к так называемому дискретному преобразованию Фурье (Discrete Fourier Transform, DFT) и обратному к нему преобразованию, которые выполняются за* $\Theta(n \lg n)$ *операций.*

Этот план действий изображен на рис. 32.1. Как мы уже говорили, при умножении двух многочленов степени меньше n получается многочлен степени меньше $2n$*, поэтому для начала мы дополняем многочлены-сомножители нулевыми коэффициентами старших степеней. После этого мы имеем дело с многочленами степени меньше* $2n$*, и потому используем "комплексные корни степени* $2n$ *из единицы", обозначаемые* ω_{2n}^{i} *при* $i = 0, 1, \ldots, 2n-1$.

Итак, повторим еще раз, как умножать два многочлена $A(x)$ *и* $B(x)$ *степени меньше n. Мы предполагаем, что n является степенью двойки — этого всегда легко достичь, добавляя нулевые коэффициенты старших степеней.*

1. Удвоение количества коэффициентов. Дополнить вектора коэффициентов, задающие многочлены $A(x)$ *и* $B(x)$*, нулевыми коэффициентами старших степеней так, чтобы в векторах коэффициентов стало по* $2n$ *элементов.*

2. Вычисление значений. При помощи быстрого преобразования Фурье (применяемого дважды — для A и для B) вычислить значения многочленов $A(x)$ *и* $B(x)$ *в точках, являющихся корнями* *степени $2n$ из единицы.*

3. Поточечное умножение. Поточечно умножить полученные значения многочленов $A(x)$ и $B(x)$ друг на друга. В результате получаются значения многочлена $C(x) = A(x)B(x)$ в корнях степени $2n$ из единицы.

4. Интерполяция. Получить коэффициенты многочлена $C(x)$ при помощи обратного быстрого преобразования Фурье, применённого его значениям в корнях из единицы.

Шаги 1 и 3 требуют времени $\Theta(n)$, а шаги 2 и 4 — времени $\Theta(n \lg n)$, как мы увидим в следующем разделе. Тем самым будет доказана следующая теорема:

Теорема 32.2

Произведение двух многочленов степени меньше n может быть вычислено за $\Theta(n \lg n)$ операций. (Многочлены на входе и выходе задаются векторами коэффициентов.)

Упражнения

32.1-1

Найдите произведение многочленов $A(x) = 7x^3 - x^2 + x - 10$ и $B(x) = 8x^3 - 6x + 3$, используя формулу (32.2).

32.1-2

Значение многочлена $A(x)$ степени меньше n в данной точке x_0 можно найти, разделив $A(x)$ на многочлен $(x - x_0)$ и получив частное $q(x)$ (которое является многочленом степени меньше $n-1$) и остаток r, для которых

$$A(x) = q(x)(x - x_0) + r.$$

Ясно, что $A(x_0) = r$. Покажите, как вычислить остаток r и коэффициенты многочлена $q(x)$ за время $\Theta(n)$, если даны x_0 и коэффициенты многочлена A.

32.1-3

Многочлен $A(x) = \sum_{j=0}^{n-1} a_j x^j$ задан набором своих значений в n отличных от 0 точках. Укажите n пар аргумент-значение, задающих многочлен $A^{\mathrm{rev}}(x) = \sum_{j=0}^{n-1} a_{n-1-j} x^j$. (Другими словами, надо указать n различных точек и найти значения многочлена A^{rev} в этих точках.)

32.1-4

Покажите, как использовать формулу (32.5) для интерполяции за время $\Theta(n^2)$. (Указание: Сначала вычислите $\prod_k (x - x_k)$, для вычисления j-ого слагаемого надо разделить полученный многочлен на $(x - x_j)$. См. упражнение 32.1-2.)

32.1-5

Почему не удаётся делить многочлены тем же методом, деля их значения поточечно? Рассмотрите отдельно случаи, когда многочлены делятся друг на друга нацело и и когда имеется остаток.

32.1-6

Рисунок 31.3 32.2 Значения $\omega_8^0, \omega_8^1, \ldots, \omega_8^7$ на комплексной плоскости, где $\omega_8 = e^{2\pi i/8}$ — главное значение корня степени 8 из единицы.

Рассмотрим два множества, A и B, каждое из которых содержит n целых чисел в диапазоне от 0 до $10n$. Мы хотим найти их декартову сумму *(Cartesian sum) A и B, определяемую как*

$$C = \{x + y : x \in A \text{ и } y \in B\}.$$

Заметьте, что C может содержать целые числа от 0 до $20n$. Нас интересуют элементы множества C, а также их "кратности" (сколькими способами данный элемент можно получить, сложив элемент A с элементом B). Покажите, что эта задача может быть решена за время $\Theta(n \lg n)$. (Указание: представьте A и B многочленами степени не больше $10n$.)

32.2 Дискретное преобразование Фурье. Быстрый алгоритм

В разделе 32.1 мы собирались использовать комплексные корни из единицы как точки, в которых вычисляются значения многочлена, и говорили, что вычисление значений в них и интерполяция проводятся за время $O(n \lg n)$. Мы сейчас объясним, как это делается с использованием дискретного преобразования Фурье и быстрого алгоритма его выполнения.

Комплексные корни единицы

Комплексным корнем степени n из единицы (complex nth root of unity) называют такое комплексное число ω, что

$$\omega^n = 1.$$

Имеется ровно n комплексных корней степени n из единицы. Они имеют вид $e^{2\pi ik/n}$ для $k = 0, 1, \ldots, n-1$. Вспоминая формулу для экспоненты комплексного числа, можно написать

$$e^{iu} = \cos u + i \sin u.$$

На рисунке 32.2 видно, что комплексные корни единицы равномерно распределены на окружности единичного радиуса с центром в нуле. Значение

$$\omega_n = e^{2\pi i/n} \tag{32.6}$$

называется главным значением корня степени n из единицы *(the principal nth root of unity). Остальные корни из единицы являются его степенями.*

Комплексные корни степени n из единицы образуют группу по умножению (см. раздел 33.3). Эта группа имеет ту же структуру, что и аддитивная группа $(\mathbb{Z}_n, +)$, поскольку равенство $\omega_n^n = \omega_n^0 = 1$ показывает, что $\omega_n^j \omega_n^k = \omega_n^{j+k} = \omega_n^{(j+k) \bmod n}$. Аналогично, $\omega_n^{-1} = \omega_n^{n-1}$. Докажем некоторые свойства корней из единицы.

Лемма 32.3 (Лемма о сокращении)

Для любых целых $n \geqslant 0$, $k \geqslant 0$ и $d > 0$

$$\omega_{dn}^{dk} = \omega_n^k. \tag{32.7}$$

Доказательство

Согласно (32.6),

$$\omega_{dn}^{dk} = (e^{2\pi i/dn})^{dk} = (e^{2\pi i/n})^k = \omega_n^k.$$

Следствие 32.4

Для любого чётного $n > 0$

$$\omega_n^{n/2} = \omega_2 = -1.$$

Доказательство

оставляется читателю в качестве упр. 32.2-1.

Лемма 32.5 (Лемма о делении пополам)

Если $n > 0$ чётно, то, возведя в квадрат все n комплексных корней степени n из единицы, мы получим все $n/2$ комплексных корней степени $n/2$ из единицы (каждый — по два раза).

Доказательство

Легко проверить, что корни ω_n^k и $\omega_n^{k+n/2}$ отличаются знаком и при возведении в квадрат дают одно и то же число $\omega_n^{2k} = \omega_{n/2}^k$.

Лемма о делении пополам позволит свести вычисление значений многочлена в корнях из единицы степени n к вычислению значений других многочленов в корнях из единицы степени $n/2$.

Лемма 32.6 (Лемма о сложении)

Для любого целого $n \geqslant 1$ и неотрицательного целого k, не кратного n, выполнено равенство

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0.$$

Доказательство

По формуле (3.3) (которая верна и для комплексных чисел) имеем

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = \frac{(\omega_n^k)^n - 1}{\omega_n^k - 1} = \frac{(\omega_n^n)^k - 1}{\omega_n^k - 1} = \frac{(1)^k - 1}{\omega_n^k - 1} = 0.$$

(знаменатель не обращается в нуль, так как k не кратно n).

Дискретное преобразование Фурье

Вспомним, что мы хотим вычислить значение многочлена

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

степени меньше n в корнях степени n из единицы, то есть в точках $\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1}$. (Напомним, что для умножения двух многочленов степени меньше m мы использовали их значения в $2m$ точках, так что нынешнее значение n удвоено по сравнению с разделом 32.1). Мы предполагаем, что n является степенью 2 (этого для наших целей достаточно, так как к многочлену всегда можно добавить нулевые старшие коэффициенты). Итак, нам задан вектор коэффициентов $a = (a_0, a_1, \ldots, a_{n-1})$ и надо вычислить

$$y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj}. \tag{32.8}$$

для $k = 0, 1, \ldots, n-1$.

Вектор $y = (y_0, y_1, \ldots, y_{n-1})$ называется дискретным преобразованием Фурье (Discrete Fourier Transform, DFT) *вектора $a = (a_0, a_1, \ldots, a_{n-1})$. Обозначение: $y = \mathrm{DFT}_n(a)$.*

Быстрое преобразование Фурье

(Fast Fourier Transform, FFT) представляет собой метод быстрого вычисления дискретного преобразования Фурье, использующий свойства комплексных корней из единицы и требующий времени $\Theta(n \lg n)$ (а не $\Theta(n^2)$, как получится при использовании формул (32.8)).

Быстрое преобразование Фурье использует метод "разделяй и властвуй". Выделим в многочлене A отдельно члены чётных и нечётных степеней, записав

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2), \tag{32.9}$$

где

$$A^{[0]}(x) = a_0 + a_2 x + a_4 x^2 + \ldots + a_{n-2} x^{n/2-1},$$

и

$$A^{[1]}(x) = a_1 + a_3 x + a_5 x^2 + \ldots + a_{n-1} x^{n/2-1}.$$

Тем самым задача вычисления $A(x)$ в точках ω_n^0, ω_n^1, …, ω_n^{n-1} сводится к

1. *вычислению значений многочленов $A^{[0]}$ и $A^{[1]}$ степени меньше $n/2$ в точках*

$$(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2; \tag{32.10}$$

2. *комбинации результатов по формуле (32.9).*

Лемма о делении пополам гарантирует, что список (32.10) содержит всего $n/2$ различных чисел, а именно, комплексных корней степени $n/2$ из единицы (каждый входит дважды). Таким образом, нам нужно вычислить значения многочленов $A^{[0]}$ и $A^{[1]}$ (степени меньше $n/2$) в $n/2$ комплексных корнях степени $n/2$ из единицы. Эти подзадачи имеют тот же вид, что исходная, но вдвое меньший размер. Мы приходим к такому рекурсивному алгоритму:

вычисления преобразования Фурье вектора $a = (a_0, a_1, \ldots, a_{n-1})$ *(где* n *— степень* 2*):*

```
\textsc{Recursive-FFT}$(a)$       % Рекурсивное БПФ

1 $n\leftarrow length[a]$ \qquad $n$ --- степень $2$
2 if $n=1$
3 \quad then return $a$
4 $\omega_n\leftarrow e^{2\pi i/n}$
5 $\omega\leftarrow 1$
6 $a^{[0]}\leftarrow (a_0,a_2,\ldots,a_{n-2})$
7 $a^{[1]}\leftarrow (a_1,a_3,\ldots,a_{n-1})$
8 $y^{[0]}\leftarrow$ \textsc{Recursive-FFT}$(a^{[0]})$
9 $y^{[1]}\leftarrow$ \textsc{Recursive-FFT}$(a^{[1]})$
10 for $k\leftarrow0$ to $n/2-1$
11 \quad do $y_k\leftarrow y_k^{[0]} + \omega y_k^{[1]}$
12 \qquad $y_{k+(n/2)}\leftarrow y_k^{[0]} - \omega y_k^{[1]}$
13 \qquad $\omega\leftarrow\omega\omega_n$
14 return $y$
```

Процедура Recursive-FFT *работает следующим образом. Строки 2–3 образуют "базис рекурсии": дискретным преобразованием Фурье для вектора длины* 1 *является сам этот вектор, так как*

$$y_0 = a_0\omega_1^0 = a_0 1 = a_0.$$

В строках 6–7 формируются вектора коэффициентов многочленов $A^{[0]}$ *и* $A^{[1]}$*. Строки 4, 5 и 13 гарантируют, что* $\omega = \omega_n^k$ *в момент выполнения строк 11–12 (мы экономим время, не вычисляя значение* ω_n^k *каждый раз заново) В строках 8–9 рекурсивно вычисляются значения*

$$y_k^{[0]} = A^{[0]}(\omega_{n/2}^k),$$

$$y_k^{[1]} = A^{[1]}(\omega_{n/2}^k),$$

или (поскольку $\omega_{n/2}^k = \omega_n^{2k}$ *по лемме о сокращении)*

$$y_k^{[0]} = A^{[0]}(\omega_n^{2k}),$$

$$y_k^{[1]} = A^{[1]}(\omega_n^{2k}).$$

В строках 11–12 собираются вместе результаты рекурсивных вычислений $\mathrm{DFT}_{n/2}$*. В строке 11 для* $y_0, y_1, \ldots, y_{n/2-1}$ *получается*

$$y_k = y_k^{[0]} + \omega_n^k y_k^{[1]} = A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) = A(\omega_n^k);$$

последнее равенство следует из (32.9). Для $y_{n/2}, y_{n/2+1}, \ldots, y_{n-1}$ *строка 12 даёт (при* $k = 0, 1, \ldots, n/2 - 1$*)*

$$\begin{aligned} y_{k+(n/2)} &= y_k^{[0]} - \omega_n^k y_k^{[1]} \\ &= y_k^{[0]} + \omega_n^{k+(n/2)} y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}) \\ &= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k+n}) \\ &= A(\omega_n^{k+(n/2)}). \end{aligned}$$

Второе равенство верно, поскольку $\omega_n^{k+(n/2)} = -\omega_n^k$*. Четвёртое равенство верно, поскольку из* $\omega_n^n = 1$ *следует, что* $\omega_n^{2k} = \omega_n^{2k+n}$*. Последнее равенство следует из уравнения (32.9). Таким образом, вектор* y*, возвращаемый процедурой* RECURSIVE-FFT*, действительно есть дискретное преобразование Фурье входного вектора* a*.*

Сколько времени занимает процедура RECURSIVE-FFT*? Если не учитывать рекурсивные вызовы, она требует времени* $\Theta(n)$*, где* n *— длина входного вектора. С учётом рекурсивных вызовов получаем такое соотношение для времени* $T(n)$ *работы процедуры*

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n).$$

Таким образом, приведённый алгоритм (рекурсивный вариант быстрого преобразования Фурье) позволяет вычислить значения многочлена степени меньше n *в комплексных корнях степени* n *из единицы за время* $\Theta(n \lg n)$*.*

Интерполяция по значениям в корнях из единицы

Нам осталось показать, как перейти обратно от значений многочлена в комплексных корнях из единицы к его коэффициентам. Для этого мы представим преобразование Фурье как умножение на матрицу, и найдём обратную матрицу.

Согласно уравнению (32.4), дискретное преобразование Фурье можно записать как матричное умножение $y = V_n a$*, где* V_n *— это матрица Вандермонда, составленная из степеней* ω_n*:*

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \ldots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \ldots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \ldots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \ldots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

Элемент матрицы V_n с индексами (k, j) равен ω_n^{kj} (при $j, k = 0, 1, \ldots, n-1$); показатели степеней в матрице V_n образуют "таблицу умножения".

Обратная операция $a = \mathrm{DFT}_n^{-1}(y)$ состоит в умножении матрицы V_n^{-1} (обратной к V_n) на y.

Теорема 32.7

Элемент с индексами (j, k) матрицы V_n^{-1} равен ω_n^{-kj}/n.

Доказательство

Покажем, что $V_n^{-1}V_n = I_n$, где I_n — это единичная $(n \times n)$-матрица. По определению, (j, j')-й элемент произведения $V_n^{-1}V_n$ есть

$$[V_n^{-1}V_n]_{jj'} = \sum_{k=0}^{n-1} (\omega_n^{-kj}/n)(\omega_n^{kj'}) = \sum_{k=0}^{n-1} \omega_n^{k(j'-j)}/n.$$

Последняя сумма равна 1 при $j = j'$ и равна 0 при $j \neq j'$ по лемме о сложении (лемма 32.6). В самом деле, $-(n-1) \leqslant j' - j \leqslant n - 1$, так что $j - j'$ не делится на n при $j \neq j'$.

Зная обратную матрицу V_n^{-1}, мы можем найти $a = \mathrm{DFT}_n^{-1}(y)$ по формуле

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k \omega_n^{-kj}, \qquad 32.11$$

для $j = 0, 1, \ldots, n-1$. Сравнив формулы (32.8) и (32.11), мы видим, что для вычисления обратного преобразования Фурье можно применить тот же алгоритм (быстрого преобразования Фурье), поменяв в нём a и y местами, заменив ω_n на ω_n^{-1} и разделив каждый элемент результата на n (см. упр. 32.2-4). Таким образом, DFT_n^{-1} также можно вычислить за время $\Theta(n \lg n)$.

Итак, мы научились переходить от коэффициентов многочлена к набору его значений в корнях из единицы и обратно за время $\Theta(n \lg n)$. Это умение можно использовать для умножения многочленов. Сформулируем соответствующий результат в виде теоремы:

Теорема 32.8 (о свёртке)

Для любых векторов a и b размерности n, где n — степень двойки выполнено равенство

$$a \otimes b = \mathrm{DFT}_{2n}^{-1}(\mathrm{DFT}_{2n}(a) \bullet \mathrm{DFT}_{2n}(b)),$$

если дополнить векторы a и b нулевыми элементами до длины $2n$, и через $\bullet$ обозначить покомпонентное произведение двух $2n$-элементных векторов.

Упражнения

32.2-1

Докажите следствие 32.4

32.2-2

Найдите дискретное преобразование Фурье вектора $(0, 1, 2, 3)$.

32.2-3

Выполните упр. 32.1-1, используя схему быстрого (время $\Theta(n \lg n)$*) алгоритма умножения многочленов (рис. 32.1).*

32.2-4

Напишите процедуру вычисления DFT_n^{-1} *за время* $\Theta(n \lg n)$.

32.2-5

Как быстро выполнить преобразования Фурье, если n *является степенью тройки? Напишите рекуррентное соотношение для времени работы соответствующей процедуры и решите его.*

*32.2-6**

Преобразование Фурье вектора из n *элементов можно вычислять не над полем* $\mathbb{C}$, *а в кольце* $\mathbb{Z}_m$ *целых чисел по модулю* m, *где* $m = 2^{tn/2} + 1$, *где* t *— произвольное положительное целое число. При этом роль* ω_n *играет* 2^t. *Покажите, что дискретное преобразование Фурье и обратное к нему для векторов с элементами из этого кольца корректно определены и могут быть вычислены за время* $O(n \lg n)$ *по той же схеме.*

32.2-7

Покажите как для данного списка чисел $z_0, z_1, \ldots, z_{n-1}$ *(не обязательно различных) найти коэффициенты многочлена* $P(x)$ *степени меньше* n, *имеющего указанные в списке корни (с учётом кратности). Время работы должно быть* $O(n \lg n)$. *(Указание. Многочлен* $P(x)$ *обращается в нуль в точке* a *тогда и только тогда, когда делится на* $(x - a)$.*)*

*32.2-8**

Рассмотрим преобразование вектора $a = (a_0, a_1, \ldots, a_{n-1})$, *переводящее его в вектор* $y = (y_0, y_1, \ldots, y_{n-1})$, *заданный формулой* $y_k = \sum_{j=0}^{n-1} a_j z^{jk}$, *где* z *— произвольное комплексное число. (Дискретное преобразование Фурье является частным случаем для* $z = \omega_n$.*) Докажите, что это преобразование можно выполнить за время* $O(n \lg n)$ *для любого комплексного числа* z. *(Указание: используйте равенство*

$$y_k = z^{k^2/2} \sum_{j=0}^{n-1} (a_j z^{j^2/2})(z^{-(k-j)^2/2}),$$

чтобы представить результат преобразования как свёртку.)

32.3 Эффективные реализации быстрого преобразования Фурье

Дискретное преобразование Фурье часто встречается на практике, причём в ситуациях, где скорость работы очень важна (обработка сигналов и т.п.). В этом разделе рассматриваются две эффективные реализации быстрого преобразования Фурье. Сначала мы рассмотрим итеративный алгоритм быстрого преобразования Фурье, который имеет лучшие константы в асимптотической оценке $O(n \lg n)$, *чем рекурсивный алгоритм раздела 32.2.*

Рисунок 31.4 32.3 Преобразование бабочки. Слева поступают два входных значения, ω_n^k умножается на $y_k^{[1]}$, справа выдаются значения суммы и разности. Рисунок можно рассматривать как схему, составленную из элементов сложения, вычитания и умножения.

Рисунок 31.5 32.4 Дерево входных векторов для рекурсивных вызовов процедуры Recursive-FFT. Исходным является вызов при $n = 8$.

Затем мы используем те же идеи для быстрой параллельной реализации.

Итеративная реализация быстрого преобразования Фурье

Прежде всего заметим, что в цикле **for** *в строках 10–13 процедуры* Recursive-FFT *дважды вычисляется значение* $\omega_n^k y_k^{[1]}$. *Введя временную переменную* t, *можно запомнить значение этого* общего подвыражения *(common subexpression — терминология, используемая разработчиками компиляторов), чтобы не вычислять его вторично:*

```
for $k \leftarrow 0$ to $n/2-1$
\quad do $t \leftarrow \omega y_k^{[1]}$
\qquad $y_k \leftarrow y_k^{[0]} + t$
\qquad $y_{k+(n/2)} \leftarrow y_k^{[0]} - t$
\qquad $\omega \leftarrow \omega \omega_n$
```

Выполняемые в этом цикле действия показаны на рис. 32.3; их называют преобразованием бабочки *(butterfly operation).*

Сейчас мы покажем, как построить итеративный (а не рекурсивный) вариант быстрого преобразования Фурье. На рисунке 32.4 показаны входные векторы в дереве рекурсивных вызовов процедуры Recursive-FFT, *начиная с ее вызова для* $n = 8$.

Указаны входные векторы для рекурсивных вызовов процедуры Recursive-FFT; *каждый такой вызов порождает два новых (для векторов половинной длины). На рисунке левый потомок соответствует первому из них, а правый — второму.*

Эти рекурсивные вызовы можно заменить вычислением снизу вверх. Расположим элементы исходного вектора a *в том порядке, в каком они появляются в листьях дерева (нижняя строка). Затем возьмём пары элементов, вычислим для них дискретное преобразование Фурье, используя преобразование бабочки. Получится содержимое второй снизу строки рисунка. (Элементы исходного вектора больше не нужны, так что можно записать результаты преобразований на их место.) Теперь из каждых двух пар мы собираем четвёрку, дважды выполнив преобразование бабочки, получается* $n/4$ *четвёрок. Далее из четвёрок собираются восьмёрки и т.д.; на последнем шаге из двух векторов длины* $n/2$, *являющихся преобразованиями Фурье чётной и нечётной части вектора* a, *из-*

готавливается преобразование Фурье всего вектора a.

Соответствующая программа использует вектор $A[0..n-1]$ в который в начале работы мы помещаем элементы вектора a в том порядке, в котором идут листья дерева рис. 32.4. (Что это за порядок, мы обсудим дальше.) Введём также переменную s, в которой будет содержаться текущий номер уровня в дереве рекурсии (растущий от $s = 1$ для листьев до $\lg n$ для корня, когда мы получаем n-элементный результат). Вот схема программы:

```
1 for $s \leftarrow 1$ to $\lg n$
2 \quad do for $k \leftarrow 0$ to $n-1$ by $2^s$
3 \qquad do преобразовать два $2^{s-1}$-элементных куска
\quad\qquad $A[k..k+2^{s-1}-1]$ и $A[k+2^{s-1}..k+2^s-1]$
\quad\qquad в $2^s$-элементный кусок $A[k..k+2^s-1]$
```

Опишем тело цикла (строка 3) более подробно. Мы воспроизводим цикл **for** *из процедуры* Recursive-FFT, *используя вместо $y^{[0]}$ кусок $A[k..k+2^{s-1}-1]$, а вместо $y^{[1]}$ — кусок $A[k+2^{s-1}..k+2^s-1]$. Значение ω (используемое в преобразовании бабочки) зависит от s: оно есть ω_m, где $m = 2^s$. Временная переменную u используется в преобразовании бабочки. Таким образом, развёртывая строку 3 в предыдущей процедуре, получаем такую программу:*

```
\textsc{FFT-Base}
1 $n \leftarrow length[a]$    \qquad $n$ есть степень 2
2 for $s \leftarrow 1$ to $\lg n$
3 \quad do $m \leftarrow 2^s$
4 \qquad   $\omega_m \leftarrow e^{2\pi i/m}$
5 \qquad   for $k \leftarrow 0$ to $n-1$ by m
6 \qquad\quad do $\omega \leftarrow 1$
7 \qquad\qquad   for $j \leftarrow 0$ to $m/2-1$
8 \qquad\qquad\quad do $t \leftarrow \omega A[k+j+m/2]$
9 \qquad\qquad\qquad   $u \leftarrow A[k+j]$
10 \qquad\qquad\qquad   $A[k+j] \leftarrow u + t$
11 \qquad\qquad\qquad   $A[k+j+m/2] \leftarrow u - t$
12 \qquad\qquad\qquad   $\omega \leftarrow \omega \omega_m$
```

Наконец, представим окончательную версию итеративной процедуры быстрого преобразования Фурье, в которой два внутренних цикла переставлены (экономия при вычислении ω) и использована дополнительная процедура Bit-Reverse-Copy(a, A), *копирующая элементы вектора a в массив A в нужном нам порядке.*

```
\textsc{Iterative-FFT}$(a)$
1 \textsc{Bit-Reverse-Copy}$(a, A)$
2 $n \leftarrow length[a]$  \quad $n$ --- степень $2$
3 for $s \leftarrow 1$ to $\lg n$
```

```
\quad do $m \leftarrow 2^s$
\qquad   $\omega_m \leftarrow e^{2\pi i/m}$
\qquad   $\omega \leftarrow 1$
\qquad   for $j \leftarrow 0$ to $m/2-1$
\qquad\quad do for $k \leftarrow j$ to $n-1$ by $m$
\qquad\qquad\quad  do $t \leftarrow \omega A [k+m/2]$
\qquad\qquad\qquad   $u \leftarrow A[k]$
\qquad\qquad\qquad   $A[k] \leftarrow u+t$
\qquad\qquad\qquad   $A[k+m/2] \leftarrow u-t$
\qquad\qquad $\omega \leftarrow \omega \omega_m$
```

Каким образом процедура Bit-Reverse-Copy *переставляет элементы входного вектора a, помещая их в массив A? Посмотрев на листья дерева на рис. 32.4, можно заметить, что они расположены ”перевёрнутом двоичном” порядке (“bit-reverse binary”). Объясним, что имеется в виду. Через* $\mathrm{rev}(k)$ *при* $k = 0, 1, 2, \ldots, 2^n - 1$ *обозначим* $(\lg n)$*-битовое целое число, двоичная запись которого получается, если написать двоичную запись числа k (всего* $\lg n$ *битов, включая начальные нули, если они есть) ”задом наперёд”. Так вот, процедура* Bit-Reverse-Copy *помещает элемент a_k в* $A[\mathrm{rev}(k)]$*. На рисунке 32.4, например, листья* $a_0, a_1, a_2, \ldots, a_7$ *появляются в позициях* 0, 4, 2, 6, 1, 5, 3, 7*; в двоичной системе номера позиций записываются как* 000, 100, 010, 110, 001, 101, 011, 111*. (Отметим, что* $\mathrm{rev}(\mathrm{rev}(k)) = k$*.)*

Чтобы понять, почему листья идут в перевёрнутом двоичном порядке, заметим, что в корне дерева налево идут индексы с нулевым младшим битом, а направо — с единичным, что после обращения битов соответствует обычному порядку (сначала числа с нулевым старшим битом, потом с единичным). Отбрасывая на каждом уровне младший бит, мы продолжаем этот процесс вниз по дереву, и в конце концов получаем перевёрнутый двоичный порядок в листьях.

Поскольку функцию $\mathrm{rev}(k)$ *легко вычислять, можно записать процедуру* Bit-Reverse-Copy *(копирование в перевёрнутом двоичном порядке) следующим образом:*

```
\textsc{Bit-Reverse-Copy}$(a,A)$
$n \leftarrow length[a]$
for $k \leftarrow 0$ to $n-1$
\quad do $A [{\mathrm rev}(k)] \leftarrow a_k$
```

Построенная итеративная реализация быстрого преобразования Фурье требует времени $\Theta(n \lg n)$*. В самом деле, вызов* Bit-Reverse-Copy *заведомо укладывается в* $O(n \lg n)$ *операций, поскольку цикл исполняется n раз, а целое число от* 0 *до* $n-1$ *занимает* $\lg n$ *битов, которые могут быть обращены за время* $O(\lg n)$*. (На практике мы обычно заранее знаем значение n, поэтому мо-*

Рисунок 31.6 32.5 Схема Parralel-FFT, вычисляющая преобразование Фурье для $n = 8$ входов. Три уровня соответствуют значениям $s = 1, 2, 3$. Для n входов подобная схема имеет глубину $\Theta(\lg n)$ и размер $\Theta(n \lg n)$.

жем изготовить таблицу значений функции $\mathrm{rev}(k)$ *заранее и пользоваться этой таблицей. Можно также использовать приём из задачи 18-1, позволяющий обратить все* n *чисел от* 0 *до* $n - 1$ *за* $O(n)$ *операций.)*

Мы должны ещё проверить, что остальная часть процедуры Iterative-FFT *требует времени* $\Theta(n \lg n)$, *Это ясно: на каждом из* $\lg n$ *уровней дерева рекурсии (рис. 32.4) имеется* n *чисел, на получение каждого из них расходуется* $O(1)$ *операций.*

Параллельная схема быстрого преобразования Фурье

Изложенные идеи позволяют построить схему из функциональных элементов, эффективно выполняющую преобразование Фурье. (Схемы из функциональных элементов описаны в главе 29; в нашем случае элементы будут выполнять арифметические операции.)

Схема, выполняющая преобразование Фурье для n *входов, которую мы назовём* Parallel-FFT, *изображена на рис. 32.5 (для* $n = 8$*). Значения на вход поступают в перевёрнутом двоичном порядке; схема содержит* $\lg n$ *каскадов, в каждом из которых параллельно выполняются* $n/2$ *преобразований бабочки Таким образом, глубина схемы есть* $\Theta(\lg n)$.

Такое распараллеливание возможно, поскольку на каждом уровне рекурсии (рис. 32.4) имеется $n/2$ *независимых преобразований бабочки, которые можно выполнять параллельно.*

Упражнения

32.3-1

Покажите работу процедуры Iterative-FFT *на примере входного вектора* $(0, 2, 3, -1, 4, 5, 7, 9)$.

32.3-2

Как реализовать алгоритм быстрого преобразования Фурье, выполняя перестановку в перевёрнутом двоичном порядке в конце, а не в начале? (Указание. Рассмотрите обратное преобразование Фурье.)

32.3-3

Сколько проводов, элементов сложения, вычитания и умножения имеется в схеме Parallel-FFT *описанной в этом разделе? (Каждый провод соединяет один выход с одним или несколькими входами.)*

*32.3-4**

Предположим, что сумматоры в схеме Parallel-FFT *иногда ломаются и выдают на выходе ноль независимо от входных значений. Предположим, что сломался ровно один сумматор, и мы не знаем, какой. Как можно быстро его найти, подавая на вход* *схемы разные наборы значений и наблюдая за выходными значениями?*

Задачи

32-1 Умножение методом разделяй и властвуй

a. Покажите, как умножить два линейных многочлена $ax+b$ и $cx+d$, сделав только три умножения. (Указание: одним из умножений будет $(a+b)\cdot(c+d)$.)

b. Предложите два алгоритма для вычисления произведения двух многочленов степени меньше n, требующих времени $\Theta(n^{\log_2 3})$. (Можно делить коэффициенты многочлена на старшую и младшую половины, а можно разделять коэффициенты при чётных и нечётных степенях.)

c. Покажите, как можно умножить два n-битных целых числа за $O(n^{\log_2 3})$ шагов, считая, что на каждом шаге обрабатывается $O(1)$ битов.

32-2 Теплицевы матрицы

Теплицевой матрицей *(Toeplitz matrix) называется матрица $A=(a_{ij})$ размера $n\times n$, для которой $a_{ij}=a_{i-1,j-1}$ для $i=2,3,\dots,n$ и $j=2,3,\dots,n$.*

a. Обязательно ли сумма двух теплицевых матриц является теплицевой? А произведение?

b. Опишите, как представить теплицевы матрицы так, чтобы две теплицевы матрицы размера $n\times n$ можно было сложить за время $O(n)$.

c. Напишите алгоритм умножения теплицевой матрицы размера $n\times n$ на вектор длины n за время $O(n\lg n)$ (используя представление пункта b).

d. Напишите эффективный алгоритм умножения двух теплицевых матриц размера $n\times n$ и оцените его время работы.

32-3 Вычисление значений всех производных многочлена в точке

Многочлен $A(x)$ степени меньше n задан своими коэффициентами $(a_0,a_1,\dots,a_{n-1})$. Мы хотим найти его значение вместе со всеми производными в заданной точке x_0.

a. Пусть известны коэффициенты $b_0,b_1,\dots,b_{n-1}$ в представлении многочлена $A(x)$ по степеням $(x-x^0)$:

$$A(x)=\sum_{j=0}^{n-1} b_j(x-x_0)^j,$$

Как тогда найти значение многочлена $A(x)$ и всех его производных в точке x_0 за время $O(n)$?

b. Объясните, как найти $b_0,b_1,\dots,b_{n-1}$ за время $O(n\lg n)$, если известны значения $A(x_0+\omega_n^k)$ при $k=0,1,\dots,n-1$.

с. Докажите, что

$$A(x_0+\omega_n^k)=\sum_{r=0}^{n-1}\left(\frac{\omega_n^{kr}}{r!}\sum_{j=0}^{n-1}f(j)g(r-j)\right),$$

где $f(j)=a_j\cdot j!$, *а*

$$g(l)=\begin{cases} x_0^{-l}/(-l)! & -(n-1)\leqslant l\leqslant 0,\\ 0 & 1\leqslant l\leqslant (n-1).\end{cases}$$

d. Объясните, как вычислить $A(x_0+\omega_n^k)$ *для* $k=0,1,\ldots,n-1$ *за время* $O(n\lg n)$. *Покажите, что все производные многочлена* $A(x)$ *в точке* x_0 *можно вычислить за время* $O(n\lg n)$.

32-4 Вычисление значений многочлена в нескольких точках

Мы видели, что значение многочлена в точке можно найти за время $O(n)$, *используя схему Горнера. Кроме того, мы знаем, что с помощью быстрого преобразования Фурье можно найти значения многочлена в* n *комплексных корнях из единицы за время* $O(n\lg n)$. *Сейчас мы покажем, как вычислить значения многочлена степени меньше* n *в* n *любых точках за время* $O(n\lg^2 n)$.

При этом мы используем (без доказательства) тот факт, что можно найти остаток от деления одного многочлена на другой за время $O(n\lg n)$. *Например, остатком от деления* $3x^3+x^2-3x+1$ *на* x^2+x+2 *равен*

$$(3x^3+x^2-3x+1)\mod(x^2+x+2)=5x-3.$$

По коэффициентам многочлена $A(x)=\sum_{k=0}^{n-1}a_kx^k$ *и* n *точкам* $x_0,x_1,\ldots,x_{n-1}$ *мы хотим найти* n *значений* $A(x_0),A(x_1),\ldots,A(x_{n-1})$. *Для* $0\leqslant i\leqslant j\leqslant n-1$ *определим многочлены* $P_{ij}(x)=\prod_{k=i}^{j}(x-x_k)$ *и* $Q_{ij}=A(x)\mod P_{ij}(x)$. *Заметьте, что степень* $Q_{ij}(x)$ *не больше* $j-i$.

а. Докажите, что $A(x)\mod(x-z)=A(z)$ *для любой точки* z.

б. Докажите, что $Q_{kk}(x)=A(x_k)$ *и что* $Q_{0,n-1}(x)=A(x)$.

в. Докажите, что $Q_{ik}(x)=Q_{ij}(x)\mod P_{ik}(x)$ *и* $Q_{kj}(x)=Q_{ij}(x)\mod P_{kj}(x)$ *для* $i\leqslant k\leqslant j$.

г. Укажите алгоритм, вычисляющий $A(x_0),A(x_1),\ldots,A(x_{n-1})$ *за время* $O(n\lg^2 n)$.

32-5 Быстрое преобразование Фурье в кольце вычетов

Дискретное преобразование Фурье (в том виде, как оно было нами определено), использует вычисления с комплексными числами, что может привести к потере точности из-за ошибок округления. Это особенно нежелательно, если исходными данными и результатами являются целые числа (например, если мы перемножаем два многочлена с целыми коэффициентами) — будет плохо, если из-за ошибок мы получим одно число вместо другого.

В этом случае можно использовать вариант преобразования Фурье, использующий кольцо вычетов (в котором вычисления проводятся точно). Мы уже видели (упр. 32.2-6), что возможно использование вычетов размером $\Omega(n)$ *битов для работы с* n*-точечным дискретным преобразованием Фурье. В этой задаче рассматривается более практичный подход, использующий* $\Omega(\lg n)$*-битовую арифметику вычетов. (мы используем материал главы 33).*

Предполагается, что число n *есть степень* 2.

a. Будем искать наименьшее k, *для которого число* $p = kn + 1$ *— простое. Найдите простой эвристический довод в пользу того, что* k *примерно равно* $\lg n$. *(Значение* k *может оказаться сильно большим или меньшим, но есть основания ожидать, что в среднем потребуется проверить* $O(\lg n)$ *кандидатов на роль* k.*)*

Пусть g *является образующей группы* $\mathbb{Z}_p^*$ *и* $w = g^k \mod p$.

b. Покажите, что дискретное преобразование Фурье и обратное к нему можно корректно определить для операций по модулю p, *используя* w *в качестве главного значения корня степени* n *из единицы.*

c. Убедитесь, что быстрое преобразование Фурье и обратное к нему можно выполнить в кольце вычетов по модулю p *за время* $O(n \lg n)$, *если считать, что операции над* $O(\lg n)$*-битовыми числами выполняются за единичное время. (Считайте* p *и* w *известными.)*

d. Вычислите дискретное преобразование Фурье вектора $(0, 5, 3, 7, 7, 2, 1, 6)$, *используя вычисления по модулю* 17. *Заметьте, что* $g = 3$ *является генератором* $\mathbb{Z}_{17}^*$.

Замечания

Пресс, Флэнери, Тьюкольски и Веттерлинг (Vetterling) [161, 162] дают хорошее описание быстрого преобразования Фурье и его приложений. Хорошее введение в теорию обработки сигналов (область, где быстрое преобразование Фурье применяется чаще всего) написали Оппенгейм и Уилски [153].

Обычно изобретение метода быстрого преобразования Фурье в 1960-х годах связывают с именами Кули и Тьюки [51]. На самом деле этот метод неоднократно встречался и раньше, но его важность стала очевидной лишь с появления современных цифровых вычислительных машин. Пресс, Флэнери, Тьюкольски и Веттерлинг указывают, что этот метод знали Рунге (Runge) и Кёниг (König) ещё в 1924 году.

33. Теоретико-числовые алгоритмы

33 Теоретико-числовые алгоритмы

Когда-то теория чисел была классическим примером красивой, но совершенно бесполезной области чистой математики. Сейчас теоретико-числовые алгоритмы широко используются — прежде всего в различных криптографических схемах, где нужны большие простые числа. В этой главе мы основные факты и алгоритмы, используемые в такого рода приложениях.

Раздел 33.1 посвящён основам теории чисел (делимость, сравнения по модулю, теорема об единственности разложения на простые множители). В разделе 33.2 рассматривается один из самых древних алгоритмов — алгоритм Евклида поиска наибольшего общего делителя двух целых чисел. В разделе 33.3 мы напоминаем основные факты арифметики в кольцах вычетов. В разделе 33.4 мы изучим остатки, даваемые кратными данного числа a *по данному модулю* n, *и научимся решать уравнение* $ax \equiv b \pmod n$ *при помощи алгоритма Евклида. Раздел 33.5 посвящён "китайской теореме об остатках". В разделе 33.6 рассматриваются остатки от деления степеней данного числа* a *по фиксированному модулю; излагается алгоритм вычисления* $a^b \bmod n$ *при помощи многократного возведения в квадрат. (Этот алгоритм играет важнейшую роль при проверке простоты чисел.) В разделе 33.7 описывается одна из так называемых криптосистем с открытым ключом — система RSA. Раздел 33.8 содержит вероятностный алгоритм проверки чисел на простоту (с его помощью ищут большие простые числа, используемые для генерации ключей в системе RSA). Наконец, в разделе 33.9 мы приведём достаточно эффективный эвристический алгоритм, помогающий разлагать на множители небольшие целые числа. Отметить, что именно отсутствие (на сегодняшний день) эффективного алгоритма разложения чисел на множители позволяет использовать систему RSA; если такой алгоритм найдётся, вся эта система рухнет в одночасье. (Так что отсутствие алгоритма может быть полезнее его существования!)*

Арифметические операции с длинными числами

Работая с большими целыми числами, мы должны догово-

риться, что считать размером входных данных той или иной задачи, а также условиться относительно стоимости элементарных арифметических операций.

В этой главе под "большим входом" (входом большого размера) мы будем понимать вход, содержащий большие числа (а не вход, содержащий много чисел — как, скажем, в задаче сортировки). Поэтому размер входа мы будем измерять количеством битов, использованных для записи чисел. Алгоритм, получающий на вход целые числа $a_1, a_2, \dots, a_k$, называется полиномиальным *(polinomial-time algorithm), если время его работы ограничено многочленом от* $\lg a_1, \lg a_2, \dots, \lg a_k$, *то есть многочленом от длин исходных данных (в двоичной системе счисления).*

До сих пор мы обычно считали, что арифметические действия (умножение, деление, вычисление остатка) выполняются за единицу времени. Это было разумно, пока не использовались большие числа, но теперь такой подход перестаёт быть адекватным. Выполнение арифметических операций само становится достаточно трудоёмким. Поэтому число операций теоретико-числового алгоритма естественнее измерять в битовых операциях *(bit operations). Например, простейший метод умножения двух $\beta-$битовых целых чисел требует $\Theta(\beta^2)$ битовых операций. Аналогичным образом деление $\beta-$битового числа на более короткое или вычисление остатка от деления $\beta-$битового числа на более короткое (стандартными методами) требуют $\Theta(\beta^2)$ битовых операций (см. упр. 33.1-11.)*

Как правило, простейшие алгоритмы не являются оптимальными. Например, несложный алгоритм, использующий метод "разделяй и властвуй", умножает два $\beta-$битовых целых числа. делая $\Theta(\beta^{\lg_2 3})$ (битовых) операций, а самый быстрый из известных на сегодняшний день алгоритмов требует $\Theta(\beta \lg \beta \lg \lg \beta)$ операций. На практике обычно применяют простейшие алгоритмы, поэтому в наших расчётах мы будем исходить из оценки оценку $\Theta(\beta^2)$ для умножения и деления.

В этой главе мы будем обращать внимание и на число арифметических, и на число битовых операций.

33.1 Начальные сведения из теории чисел

В этом разделе мы напомним некоторые свойства множеств целых чисел ($\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$) и натуральных чисел ($\mathbb{N} = \{0, 1, 2, \dots\}$).

Делимость и делители.

Говорят, что d делит *a (d divides a, запись $d \mid a$), если $a = kd$ при некотором целом k. Синонимы: " d является* делителем *a" (d*

is a divisor of a), "a делится на d". "a кратно d" (a is a multiple of d). Число 0 кратно любому числу. Если $a > 0$ и $d \mid a$, то $|d| \leqslant |a|$. Если d не делит a, пишут $d \nmid a$.

Обычно перечисляют только положительные делители; например, делители числа 24 — это числа 1, 2, 3, 4, 6, 8, 12 и само число 24.

Простые и составные числа.

Число a, не имеющее делителей, кроме ± 1 и $\pm a$, называется простым (prime); таковы числа

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, \ldots .$$

Простых чисел бесконечно много (упр. 33.1-1). Целое число $a > 1$, не являющееся простым, называется составным (composite). Число 1, число 0, а также отрицательные целые числа мы не относим ни к простым, ни к составным.

Деление с остатком. Сравнения по модулю.

Фиксируем целое число n. Все целые числа делятся на n групп в зависимости от остатков, которые они дают при делении на n: числа вида kn (кратные n) образуют одну группу, числа вида $kn + 1$ — другую и т.п.

Строго говоря, тут следует сослаться на такую теорему (подробности можно найти в учебнике по теории чисел, например, в книге Нивена и Цукермана [151]):

Теорема 33.1 (о делении с остатком)

Для любого целого числа a и положительного целого числа n существует единственная пара целых числа q и r, для которых $0 \leqslant r < n$ и $a = qn + r$.

Число $q = \lfloor a/n \rfloor$ называется частным (quotient); число r, обозначаемое $a \bmod n$, называется остатком (remainder, residue). Очевидно, что $n \mid a$ тогда и только тогда, когда $a \bmod n = 0$, что

$$a = \lfloor a/n \rfloor n + (a \bmod n) \tag{33.1}$$

и что

$$a \bmod n = a - \lfloor a/n \rfloor n. \tag{33.2}$$

Говорят, что a сравнимо с b по модулю n (a is equivalent to b, modulo n; запись: $a \equiv b \pmod{n}$) если $(a \bmod n) = (b \bmod n)$, то есть $n \mid (b - a)$. Если a не сравнимо с b по модулю n, пишут $a \not\equiv b \pmod{n}$. Например, $61 \equiv 6 \pmod{11}$ и $-13 \equiv 22 \equiv 2 \pmod{5}$.

Все целые числа делятся на n классов эквивалентности по модулю n (eqivalence classes modulo n). Число a входит в класс

$$[a]_n = \{a + kn : k \in \mathbf{Z}\}.$$

Например, $[3]_7 = [-4]_7 = [10]_7 = \{\ldots, -11, -4, 3, 10, 17, \ldots\}$. Множество всех классов эквивалентности по модулю n обозначается

$\mathbb{Z}_n$:

$$\mathbb{Z}_n = \{[a]_n : 0 \leqslant a \leqslant n-1\}. \tag{33.3}$$

Выбирая в каждом классе по представителю, можно считать, что

$$\mathbb{Z}_n = \{0, 1, \ldots, n-1\}. \tag{33.4}$$

Общие делители. Наибольший общий делитель.

Среди всех общих делителей *(common divisors) данных чисел a и b можно выбрать* наибольший общий делитель. *(greatest common divisor), который обозначают* $\gcd(a, b)$ *или* (a, b) *(он определён, если хотя бы одно из чисел a и b отлично от* 0*; положим для удобства* $\gcd(0, 0) = 0$.

Важные свойства общих делителей и наибольшего общего делителя:

$$\text{если } d \mid a \text{ и } d \mid b, \text{ то } d \mid (a+b) \text{ и } d \mid (a-b). \tag{33.5}$$

$$\text{если } d \mid a \text{ и } d \mid b, \text{ то } d \mid (ax+by), \tag{33.6}$$

при любых целых x и y. Если $a \mid b$ то $|a| \leqslant |b|$ или $b = 0$, поэтому

$$\text{если } a \mid b \text{ и } b \mid a, \text{ то } a = \pm b. \tag{33.7}$$

$$gcd(a, b) = gcd(b, a) \tag{33.8}$$

$$gcd(a, b) = gcd(-a, b), \tag{33.9}$$

$$gcd(a, b) = gcd(|a|, |b|), \tag{33.10}$$

$$gcd(a, 0) = |a|, \tag{33.11}$$

$$gcd(a, ka) = |a| \text{ при всяком } k \in \mathbf{Z}. \tag{33.12}$$

Теорема 33.2

Наибольший общий делитель целых чисел a и b, не равных 0 одновременно, является наименьшим положительным элементом множества $\{ax + by : x, y \in \mathbf{Z}\}$ целочисленных линейных комбинаций чисел a и b.

Доказательство. Пусть наименьший положительный элемент этого множества равен s. Тогда $s = ax + by$ для некоторых $x, y \in \mathbb{Z}$. Положим $q = \lfloor a/s \rfloor$. Согласно (33.2),

$$\begin{aligned} a \bmod s &= a - qs \\ &= a - q(ax + by) \\ &= a(1 - qx) + b(-qy), \end{aligned}$$

и потому остаток от деления a на s тоже является линейной комбинацией a и b. Но s был наименьшей положительной комбинацией такого вида, так что $a \bmod s = 0$ *и $s \mid a$. По аналогичным* *причинам верно* $s \mid b$. *Таким образом,* s *является общим делителем* a *и* b, *поэтому* $gcd(a,b) \geqslant s$. *С другой стороны,* $gcd(a,b)$ *делит как* a, *так и* b, *а потому делит и* $s = ax + by$ *(33.6). Поскольку* $s > 0$, *отсюда следует, что* $gcd(a,b) \leqslant s$. *Таким образом,* $gcd(a,b) \geqslant s$ *и* $gcd(a,b) \leqslant s$, *так что* $gcd(a,b) = s$,

Следствие 33.3

Наибольший общий делитель двух целых чисел кратен любому их общему делителю.

Доказательство.

По теореме 33.2 $gcd(a,b)$ *является линейной комбинацией* a *и* b.

Следствие 33.4

Для любых целых чисел a *и* b *и неотрицательного целого числа* n *выполняется соотношение* $gcd(an, bn) = ngcd(a,b)$.

Доказательство. Элементы вида $anx + bny$ *получаются из элементов вида* $ax + by$ *умножением на* n, *так что это относится и к наименьшему элементу такого вида.*

Следствие 33.5

Для любых положительных целых чисел n, a *и* b *из* $n \mid ab$ *и* $gcd(a,n) = 1$ *следует* $n \mid b$.

Доказательство

оставляется читателю (упр. 33.1-4).

Взаимно простые числа.

Целые числа a *и* b взаимно просты *(are relatively prime), если* $gcd(a,b) = 1$.

Теорема 33.6

Если $gcd(a,p) = 1$ *и* $gcd(b,p) = 1$, *то* $(ab,p) = 1$ *(для любых целых чисел* a, b, p*).*

Доказательство. По теореме 33.2 найдутся целые числа x, y, x' *и* y', *для которых*

$$\begin{aligned} ax + py &= 1, \\ bx' + py' &= 1. \end{aligned}$$

Перемножая эти равенства, мы видим, что

$$ab(xx') + p(ybx' + y'ax + pyy') = 1,$$

так что 1 *является целочисленной линейной комбинацией* ab *и* p; *осталось сослаться на теорему 33.2.*

Целые числа $n_1, n_2, \ldots, n_k$ *называются* попарно взаимно простыми *(pairwise relatively prime), если любые два из них взаимно просты.*

Разложение на простые множители

Теорема 33.7

Если простое число p *делит произведение двух целых чисел* a *и* b, *то* $p \mid a$ *или* $p \mid b$.

Доказательство. Если это не так и p не делит ни a, ни b, то P взаимно просто с этими числами (других делителей у p нет), и потому взаимно просто с их произведением (теорема 33.6).

Теорема 33.8 (Существование и единственность разложения)

Всякое составное число a единственным образом представляется в виде

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r},$$

где $p_1 < p_2 < \cdots < p_r$ — простые числа, а e_i — положительные целые числа.

Доказательство оставляется читателю (упр. 33.1-10).

Упражнения.

33.1-1

Докажите, что существует бесконечно много простых чисел. (Указание: ни одно из простых чисел $p_1, p_2, \ldots, p_k$ не делит число $(p_1 p_2 \cdots p_k) + 1$.)

33.1-2

Докажите, что если $a \mid b$ и $b \mid c$, то $a \mid c$.

33.1-3

Пусть число p — простое, а $0 < k < p$. Докажите, что $(k, p) = 1$.

33.1-4

Докажите следствие 33.5.

33.1-5

Пусть число p — простое, а $0 < k < p$. Докажите, что $p \mid C_p^k$. Выведите отсюда, что для любых целых a, b и простого p выполняется сравнение

$$(a + b)^p \equiv a^p + b^p \pmod{p}.$$

33.1-6

Пусть a и b — целые числа, причём $a \mid b$ и $b > 0$. Докажите, что $(x \bmod b) \bmod a = x \bmod a$ при любом целом x. Докажите, что (при тех же допущениях) из $x \equiv y \pmod{b}$ следует, что $x \equiv y \pmod{a}$ при любых целых x и y.

33.1-7

Фиксируем целое положительное k. Будем называть целое число n (точной) k-й степенью *(kth power), если n равно a^k при некотором целом a. Целое число $n > 1$ назовём* нетривиальной степенью *(nontrivial power), если оно является k-й степенью при некотором целом $k > 1$. Предложите полиномиальный алгоритм, определяющий, является ли данное число n нетривиальной степенью.*

33.1-8

Докажите свойства (33.8)–(33.12).

33.1-9

Докажите, что функция gcd ассоциативна, то есть для $gcd(a, gcd(b, c)) = gcd(gcd(a, b), c)$ для любых целых a, b, c.

*33.1-10**

Докажите теорему 33.8.

33.1-11

Предложите алгоритмы, вычисляющие за время $O(\beta^2)$ частное и остаток от деления β-битового целого числа на более короткое (временем считаем число битовых операций).

33.1-12

Постройте эффективный алгоритм, переводящий β-битовое целое число из двоичной системы в десятичную. Покажите, что если умножение и деление двух целых чисел длины не более β (имеется в виду длина двоичной записи) требует времени $M(\beta)$, то перевод β−битового целого числа из двоичной системы в десятичную может быть выполнен за время $\Theta(M(\beta)\lg\beta)$.

(Указание. Используйте метод "разделяй и властвуй", получая отдельно левую и правую части результата.)

33.2 Наибольший общий делитель

В этом разделе мы предполагаем, что все рассматриваемые целые числа положительны (для наибольшего общего делителя знак роли не играет).

Можно вычислять (a, b), разлагая a и b на множители и отбирая общие множители: если

$$a = p_1^{e_1} p_2^{e_2} \cdot \ldots \cdot p_r^{e_r} \qquad (33.13)$$

и

$$b = p_1^{f_1} p_2^{f_2} \cdot \ldots \cdot p_r^{f_r}, \qquad (33.14)$$

то

$$gcd(a, b) = p_1^{\min(e_1,f_1)} p_2^{\min(e_2,f_2)} \cdot \ldots \cdot p_r^{\min(e_r,f_r)} \qquad (33.15)$$

Но разложение на множители, как мы увидим в разделе 33.9, является трудной задачей, так что его лучше избегать.

Алгоритм Евклида для нахождения наибольшего общего делителя основан на следующей теореме.

Теорема 33.9 (Рекуррентная формула для gcd)

Пусть a — целое неотрицательное, а b — целое положительное число. Тогда $gcd(a, b) = gcd(b, a \bmod b)$.

Доказательство.

Пара (a, b) имеет те же делители, что и пара $(b, a \bmod b)$ ($a \bmod b$ является целочисленной линейной комбинацией a и b и наоборот). Поэтому и наибольший общий делитель у этих пар одинаковый.

Алгоритм Евклида.

Приводимый ниже алгоритм вычисления наибольшего общего делителя описан в книге Евклида "Начала" (около 300 г. до Р.Х.), хотя, возможно, был известен и ранее. Мы запишем его в виде рекурсивной процедуры; её входом являются неотрицательные целые числа a и b.

```
Euclid (a,b)
if b=0
  then return a
  else return Euclid (b, a mod b)
```

Например, при вычислении $\text{Euclid}(30,21) = \text{Euclid}(21.9) = \text{Euclid}(9,3) = \text{Euclid}(3,0) = 3$ процедура вызывает себя трижды.

Правильность процедуры Euclid вытекает из соотношения (33.11) и из теоремы 33.9. Процедура не может работать бесконечно, поскольку рекурсивный вызов происходит с меньшим значением второго аргумента.

Время работы алгоритма Евклида.

Оценим время работы алгоритма Евклида, считая, что размера входа случай $a > b \geqslant 0$ (при $b > a \geqslant 0$ процедура $\text{Euclid}(a,b)$ первым делом вызывает себя, переставив аргументы местами, а при $a = b > 0$ работа процедуры завершается после единственного рекурсивного вызова, поскольку $a \bmod a = 0$). Заметим, что во всех рекурсивных вызовах первый аргумент будет строго больше второго (остаток меньше делителя).

Время работы процедуры Euclid пропорционально глубине рекурсии. Для её оценки нам пригодятся числа Фибоначчи F_k, определённые рекуррентным соотношением (2.13).

Лемма 33.10

Пусть $a > b \geqslant 0$. Если процедура $\text{Euclid}(a,b)$ во время работы вызывает себя k раз ($k \geqslant 1$), то $a \geqslant F_{k+2}$ и $b \geqslant F_{k+1}$.

Доказательство

проведём индукцией по k. Базисом индукции служит значение $k = 1$. Если происходит хоть один рекурсивный вызов, то $b \geqslant 1 = F_2$; по условию теоремы $a > b$, так что $a \geqslant 2 = F_3$.

Пусть лемма выполнена для случая $k-1$ вызовов; докажем её для k вызовов. На первом шаге процедура $\text{Euclid}(a,b)$ выполняет вызов $\text{Euclid}(b, a \bmod b)$, внутри которого происходит $k-1$ рекурсивных вызовов, так что по предположению индукции $b \geqslant F_{k+1}$ и $(a \bmod b) \geqslant F_k$. Из неравенства $a > b > 0$ следует, что $\lfloor a/b \rfloor \geqslant 1$ и $b + (a \bmod b) = b + (a - \lfloor a/b \rfloor b) \leqslant a$, так что $a \geqslant b + (a \bmod b) \geqslant F_{k+1} + F_k = F_{k+2}$.

Из леммы 33.10 вытекает следующая теорема.

Теорема 33.11 (Теорема Ламе)

Пусть k — целое положительное число. Если $a > b \geqslant 0$ и $b < F_{k+1}$, то процедура $\text{Euclid}(a,b)$ выполняет менее k рекурсивных

Рисунок 33.1 33.1: Работа процедуры Extended-Euclid на входе $(99, 78)$. Extended-Euclid$(99, 78)$ возвращает тройку $(3, -11, 14)$, так что $(99, 78) = 3 = 99 \cdot (-11) + 78 \cdot 14$.

вызовов.

Покажем, что оценка теоремы 33.11 неулучшаема, рассмотрев вычисление наибольшего общего делителя соседних чисел Фибоначчи. Процедура Euclid(F_3, F_2) *выполняет один рекурсивный вызов. Поскольку* $F_{k+1} \bmod F_k = F_{k-1}$, *вычисление будет идти так:* $gcd(F_{k+1}, F_k) = gcd(F_k, (F_{k+1} \bmod F_k)) = (F_k, F_{k-1})$, *поэтому вычисление* Euclid(F_{k+1}, F_k) *включает ровно* $k - 1$ *рекурсивных вызовов и верхняя оценка достигается.*

Поскольку F_k *примерно равно* $\varphi^k/\sqrt{5}$, *где* $\varphi = (1 + \sqrt{5})/2$ *— так называемое "золотое сечение" см. (2.14)), число рекурсивных вызовов для* Euclid(a, b) *(при* $a > b \geqslant 0$*) составляет* $O(\lg b)$. *(Более точную оценку см. в упр. 33.2-5.) Если процедура* Euclid *применяется к двум* β*-битовым числам, то ей приходится выполнять* $O(\beta)$ *арифметических операций, или* $O(\beta^3)$ *битовых (считаем, что умножение и деление* β*−битовых чисел требует* $O(\beta^2)$ *битовых операций). На самом деле справедлива более сильная оценка* $O(\beta^2)$ *на число битовых операций, выполняемых процедурой* Euclid *(задача 33-2).*

Расширенный алгоритм Евклида.

Немного дополнив алгоритм Евклида, можно получать с его помощью коэффициенты x *и* y, *для которых*

$$d = (a, b) = ax + by \tag{33.18}$$

(теорема 33.2). Обратите внимание, что коэффициенты x *и* y *могут оказаться не только положительными, но и нулевыми или отрицательными. (Эти коэффициенты нужны для вычисления обратных элементов в кольце вычетов.)*

Процедура Extended-Euclid *получает на вход произвольную пару целых чисел и выдаёт на выходе тройку целых чисел* (d, x, y), *удовлетворяющих соотношению (33.18).*

```
Extended-Euclid (a,b)
if b=0
 then return (a,1,0)
(d',x',y') \gets Extended-Euclid(b,a mod b)
(d,x,y) \gets (d', y', x'-\lfloor a/b\rfloor y')
return (d,x,y)
```

Рисунок 33.1 иллюстрирует работу процедуры Extended-Euclid *на входе* $(99, 78)$

Если $b = 0$, *процедура* Extended-Euclid *выдаёт значение* $d = a$, *а также значения* $x = 1$ *и* $y = 0$, *для которых* $d = a = ax + by$.

Если $b \neq 0$, *рекурсивный вызов позволяет найти числа* (d', x', y'), *таких, что* $d' = (b, a \bmod b)$ *и*

$$d' = bx' + (a \bmod b)y'. \qquad (33.19)$$

Как мы знаем, $d' = d$, *так что осталось найти* x *и* y, *для которых* $d = ax+by$. *Перепишем равенство (33.19):* $d = bx' + (a - \lfloor a/b \rfloor b)y' = ay' + b(x' - \lfloor a/b \rfloor y')$. *Таким образом, можно положить* $x = y'$ *и* $y = x' - \lfloor a/b \rfloor y'$.

Как и раньше, число рекурсивных вызовов при работе Extended-Euclid(a, b) *составляет* $O(\lg b)$.

Упражнения

33.2-1

Объясните, каким образом из равенств (33.13)–(33.14) следует равенство (33.15).

33.2-2

Проследите за выполнением процедуры Extended-Euclid *на входе* $(899, 493)$ *и найдите тройку* (d, x, y).

33.2-3

Докажите, что для любых целых чисел a, k *и* n *выполняется равенство* $gcd(a, n) = gcd(a + kn, n)$.

33.2-4

Перепишите процедуру Euclid, *заменив рекурсию циклом и ограничившись фиксированным объёмом памяти (используя лишь фиксированное число целых переменных)*

33.2-5

Пусть $a > b \geqslant 0$. *Покажите, что процедура* Euclid(a, b) *выполняет не более* $1 + \log_\varphi b$ *рекурсивных вызовов. Докажите более точную оценку* $1 + \log_\varphi(b/gcd(a, b))$.

33.2-6

Какой ответ даёт процедура Extended-Euclid(F_{k+1}, F_k)*?*

33.2-7

Докажите, что если $d \mid a$, $d \mid b$ *и* $d = ax + by$, *то* $d = gcd(a, b)$.

33.2-8

Распространим определение функции gcd на большее число аргументов, положив $gcd(a_0, a_1, \ldots, a_n) = gcd(a_0, gcd(a_1, \ldots, a_n))$. *Покажите, что результат не зависит от порядка аргументов. Предложите алгоритм нахождения коэффициентов* $x_0, x_1, \ldots, x_n$, *для которых* $gcd(a_0, a_1, \ldots, a_n) = a_0x_0 + a_1x_1 + \cdots + a_nx_n$,*, выполняющий* $O(n + \lg(\max_i a_i))$ *операций деления.*

33.2-9

Назовём наименьшим общим кратным *(least common multiple) целых чисел* $a_1, a_2, \ldots, a_n$ *наименьшее положительное целое число, кратное каждому из них (обозначение* $lcm(a_0, a_1, \ldots, a_n)$*). Укажите эффективный алгоритм, вычисляющий* $lcm(a_0, a_1, \ldots, a_n)$ *и использующий операцию поиска наибольшего общего делителя (двух чисел) в качестве подпрограммы.*

33.2-10

Докажите, что числа n_1, n_2, n_3 и n_4 попарно взаимно просты тогда и только тогда, когда $gcd(n_1n_2, n_3n_4) = gcd(n_1n_3, n_2n_4) = 1$.

Покажите, что это утверждение можно обобщить, доказать утверждение такого типа: числа $n_1, n_2, \ldots, n_k$ попарно взаимно просты, если и только если $gcd(a_1, b_1) = gcd(a_2, b_2) = \ldots = gcd(a_t, b_t) = 1$, где $t = \lceil \lg k \rceil$, а числа a_i и b_i представляют собой произведения некоторых n_i.

33.3 Модулярная арифметика

Арифметические операции по модулю n проводятся над числами $0, 1, \ldots, n-1$ так: если результат сложения, вычитания или умножения выходит за пределы указанного интервала, то он заменяется остатком при делении на n. Сложнее обстоит дело с делением. Чтобы разобраться в этом, напомним некоторые понятия теории групп.

Конечные группы.

Множество S с определённой на нём бинарной операцией $\oplus$ называется группой *(group), если выполнены такие свойства:*

1. Замкнутость *(closure): $a \oplus b \in S$ для любых $a, b \in S$.*
2. *Существование* нейтрального элемента *(identity): существует элемент $e \in S$, для которого $e \oplus a = a \oplus e = a$ для любого $a \in S$ (такой элемент может быть только один, так как $e = e \cdot e' = e'$ для любых двух элементов e и e' с таким свойством).*
3. Ассоциативность *(associativity): $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ для любых $a, b, c \in S$.*
4. *Существование* обратных элементов *(inverses): для всякого $a \in S$ найдётся единственный элемент $b \in S$, для которого $a \oplus b = b \oplus a = e$.*

Например, целые числа с операцией сложения: образуют группу. В ней 0 служит нейтральным элементом, а обратным (по сложению) элементом к числу a является число $(-a)$. Группа называется абелевой *(abelian), если выполнено свойство* коммутативности*: $a \oplus b = b \oplus a$ для всех $a, b \in S$.*

Группа называется конечной *(finite), если число элементов в ней конечно.*

Аддитивные и мультипликативные группы вычетов

Можно построить две группы, элементами которых будут вычеты (остатки по модулю n, или классы эквивалентности по модулю n, см. раздел 33.1).

Мы складываем и умножаем вычеты по модулю n по правилам

$$[a]_n +_n [b]_n = [a+b]_n, \quad [a]_n \cdot_n [b]_n = [ab]_n.$$

Рисунок 33.2 33.2 (a) Группа вычетов по модулю 6 по сложению $(\mathbb{Z}_6, +_6)$. (b) Группа обратимых вычетов по модулю 15 по умножению $(\mathbb{Z}^*_{15}, \cdot_{15})$.

Эти определения корректны, так как сумма (произведение) не изменится по модулю n, если изменить слагаемые (множители) на эквивалентные по модулю n.

Аддитивная группа вычетов по модулю n *(additive group modulo n) содержит вычеты $[0]_n, [1]_n, \ldots, [n-1]_n$; они складываются описанным выше образом; она обозначается $(\mathbb{Z}_n, +_n)$.*

Теорема 33.12

Система $(\mathbb{Z}_n, +_n)$ является конечной абелевой группой.

Доказательство.

Ассоциативность и коммутативность операции $+_n$ следуют из аналогичных свойств сложения целых чисел. Нейтральным элементом является 0 (точнее говоря, $[0]_n$). Обратным (относительно групповой операции) элементом к a (точнее, $[a]_n$) служит $(-a)$ (то есть, $[-a]_n$ или $[n-a]_n$).

Несколько сложнее определяется мультипликативная группа вычетов по модулю n *(mutiplicative group modulo n). Элементы этой группы образуют множество $\mathbb{Z}^*_n$, состоящее из элементов $\mathbb{Z}_n$, взаимно простых с n. Понятие взаимной простоты имеет смысл (не зависит от выбора представителя в классе эквивалентности): если k — целое число, то $(a, n) = 1$ равносильно $(a + kn, n) = 1$ (упр. 33.2-3).*

В качестве примера рассмотрим случай $n = 15$ (рис. 33.2 (b), где перечислены элементы соответствующей группы и показана таблица умножения).

Теорема 33.13

Система $(\mathbb{Z}^*_n, \cdot_n)$ является конечной абелевой группой.

Доказательство.

Проверим, что любой элемент имеет обратный в смысле групповой операции. (Нейтральным элементом является класс $[1]$.) Чтобы найти обратный к элементу a, рассмотрим тройку (d, x, y), выдаваемую процедурой Extended-Euclid(a, n). *Поскольку $a \in \mathbf{Z}^*_n$, числа a и n взаимно просты и $d = (a, b) = 1$, поэтому $ax + ny = 1$ и $ax \equiv 1 \pmod{n}$. Таким образом, элемент $[x]_n$ является обратным к $[a]_n$ в группе $(\mathbb{Z}^*_n, \cdot_n)$. Единственность обратного можно доказать (как и для любой группы) следующим образом: если x и x' обратны к a, то $(x \oplus a) \oplus x' = e \oplus x' = x'$, а переставив скобки по ассоциативности, получим $x \oplus (a \oplus x') = x \oplus e = x$,*

В дальнейшем мы для простоты будем обозначать сложение и умножение по модулю обычными знаками $+$ и $\cdot$ (иногда опуская знак умножения), а аддитивную и мультипликативную группы вычетов по модулю n будем обозначать $\mathbb{Z}_n$ и $\mathbb{Z}^*_n$ (не упоминая *групповую операцию). Элемент, обратный (относительно операции умножения) к* a, *мы будем обозначать* $a^{-1} \bmod n$. *Как обычно, частное* a/b *в* $\mathbb{Z}_n^*$ *определяется как* $ab^{-1} \pmod{n}$. *Например, в* $\mathbb{Z}_{15}^*$ *имеем* $7^{-1} \equiv 13 \pmod{15}$, *поскольку* $7 \cdot 13 \equiv 91 \equiv 1 \pmod{15}$, *откуда* $4/7 \equiv 4 \cdot 13 \equiv 7 \pmod{15}$.

Число элементов в $\mathbb{Z}_n^*$ *обозначается* $\varphi(n)$. *Функция* φ *называется* φ-функцией Эйлера *(Euler's phi function). Можно доказать такую формулу для функции Эйлера:*

$$\varphi(n) = n\left(1 - \frac{1}{p_1}\right) \cdot \ldots \cdot \left(1 - \frac{1}{p_s}\right), \tag{33.20}$$

где $p_1, \ldots, p_s$ *— список всех простых делителей числа* n. *Можно пояснить эту формулу так: случайное число* t *взаимно просто с* n, *если оно не делится на* p_1 *(вероятность чего есть* $(1 - 1/p_1)$*), не делится на* p_2 *(вероятность* $(1 - 1/p_2)$ *и т.д., а события эти независимы.*

Например, $\varphi(45) = 45(1-1/3)(1-1/5) = 24$, *поскольку простыми делителями числа* 45 *являются числа* 3 *и* 5. *Для простого числа* p *имеем*

$$\varphi(p) = p - 1, \tag{33.21}$$

так как все числа $1, 2, \ldots, p-1$ *взаимно просты с* p. *Если число* n *составное, то* $\varphi(n) < n - 1$.

Подгруппы.

Пусть $(S, \oplus)$ *является группой, а* $S' \subseteq S$. *Если* $(S', \oplus)$ *тоже является группой, то* $(S', \oplus)$ *называют* подгруппой *(subgroup) группы* $(S, \oplus)$. *Например, чётные числа образуют подгруппу группы целых чисел (с операцией сложения).*

Теорема 33.14

Замкнутое подмножество конечной группы является подгруппой: если $(S, \oplus)$ *— конечная группой,* $S' \subset S$ *и* $a \oplus b \in S'$ *для любых* $a, b \in S'$, *то* $(S', \oplus)$ *является подгруппой группы* $(S, \oplus)$.

Доказательство

оставляется читателю (упр. 33.3-2).

Пример: множество $\{0, 2, 4, 6\} \subseteq \mathbb{Z}_8$ *замкнуто относительно сложения и образует подгруппу группы* $\mathbb{Z}_8$.

Следующая теорема накладывает на размер подгрупп важные ограничения.

Теорема 33.15 (Теорема Лагранжа)

Если $(S', \oplus)$ *является подгруппой конечной группы* $(S, \oplus)$, *то* $|S'|$ *делит* $|S|$.

Доказательство

можно найти в учебниках алгебры (группа S *разбивается на непересекающиеся классы вида* $x \oplus S'$, *каждый из которых содержит* $|S'|$ *элементов).*

Подгруппа S' *группы* S, *не совпадающая со всей группой, называется* собственной *(proper) подгруппой.*

Следствие 33.16

Если S' *является собственной подгруппой конечной группы* S, *то* $|S'| < |S| \leqslant 2$.

Это (очевидное) следствие теоремы Лагранжа будет использовано при анализе вероятностного алгоритма Миллера — Рабина (проверка простоты).

Подгруппа, порождённая элементом группы.

Пусть a *— некоторый элемент конечной группы* S. *Рассмотрим последовательность элементов*

$$e, a, a \oplus a, a \oplus a \oplus a, \ldots$$

По аналогии со степенями (групповая операция соответствует умножению) будем писать $a^{(0)} = e$, $a^{(1)} = a$, $a^{(2)} = a \oplus a$, $a^{(3)} = a \oplus a \oplus a$ *и т.д. Легко видеть, что* $a^{(i)} \oplus a^{(j)} = a^{(i+j)}$, *в частности,* $a^{(i)} \oplus a = a^{(i+1)}$. *Аналогичное утверждение можно сформулировать и для "отрицательных степеней", в частности,* $a^{(i)} \cdot a^{-1} = a^{(i-1)}$.

Если группа S *конечна, то последовательность*

$$e, a, a \oplus a, a \oplus a \oplus a, \ldots$$

будет периодична (следующий элемент определяется предыдущим, поэтому раз повторившись, элементы будут повторяться по циклу). Предпериода при этом не будет, так как каждый элемент может быть получен из следующего (применением групповой операции к нему и к a^{-1}*) и потому перед равными элементами идут равные. Таким образом, последовательность имеет вид*

$$e = a^{(0)}, a^{(1)}, a^{(2)}, \ldots, a^{(n-1)}, a^{(t)} = e, \ldots$$

(дальше всё повторяется) и содержит t *различных элементов, где* t *— наименьшее положительное число, для которого* $a^{(t)} = e$. *Это число называется* порядком *(order) элемента* a *и обозначается* $\operatorname{ord}(a)$.

Указанные n *элементов образуют подгруппу. Это следует из теоремы* 33.14, *кроме того, это можно проверить непосредственно, так как групповая операция соответствует сложению "степеней". Эта подгруппа называется* порождённой элементом a *(subgroup generated by a) и обозначается* $\langle a \rangle$ *или, если мы хоти явно указать групповую операцию,* $(\langle a \rangle, \oplus)$. *Элемент* a *называют* образующей *(generator) подгруппы* $\langle a \rangle$*; говорят, что он* порождает *(generates) эту подгруппу. Например, элемент* $a = 2$ *группы* $\mathbb{Z}_6$ *порождает подгруппу. состоящая из элементов* $0, 2, 4$.

Вот несколько подгрупп группы $\mathbb{Z}_6$, *порождённых различными элементами:* $\langle 0 \rangle = \{0\}$, $\langle 1 \rangle = \{0, 1, 2, 3, 4, 5\}$, $\langle 2 \rangle = \{0, 2, 4\}$. *Аналогичный пример для мультипликативной группы* $\mathbb{Z}_7^*$*: здесь* $\langle 1 \rangle = \{1\}$, $\langle 2 \rangle = \{1, 2, 4\}$, $\langle 3 \rangle = \{1, 2, 3, 4, 5, 6\}$.

Из сказанного нами вытекает

Теорема 33.17

Пусть $(S, \oplus)$ *— конечная группа. Если* $a \in S$, *то размер подгруппы, порождаемой* a, *совпадает с порядком* a *(то есть,* $|\langle a \rangle| = ord(a)$*).*

Следствие 33.18

Последовательность $a^{(1)}, a^{(2)}, \ldots$ *периодична с периодом* $t = ord(a)$*; иначе говоря,* $a^{(i)} = a^{(j)}$ *тогда и только тогда, когда* $i \equiv j \pmod{t}$.

Периодичность позволяет продолжить последовательность в обе стороны, определив $a^{(i)}$ *как* $a^{(i)} = a^{(i \bmod t)}$ *(ри всяком целом* i *(в том числе и отрицательном.)*

Следствие 33.19

В конечной группе $(S, \oplus)$ *с единицей* e *для всякого* $a \in S$ *выполняется равенство* $a^{(|S|)} = e$.

Доказательство

По теореме Лагранжа $ord(a) \mid |S|$, *откуда* $|S| \equiv 0 \pmod{t}$, *где* $t = ord(a)$.

Упражнения

33.3-1

Напишите таблицы для групповых операций в группах $(\mathbb{Z}_4^*, +_4)$ *и* $(\mathbb{Z}_5^*, \cdot_5)$. *Докажите, что эти группы изоморфны, то есть постройте взаимно-однозначное соответствие* α *между их элементами, удовлетворяющее следующему свойству: равенство* $a + b \equiv c \pmod{4}$ *должно выполняться одновременно с равенством* $\alpha(a) \cdot \alpha(b) \equiv \alpha(c) \pmod{5}$.

33.3-2

Докажите теорему 33.14

33.3-3

Пусть p *— простое число,* k *— положительное целое число. Докажите, что* $\varphi(p^k) = p^{k-1}(p-1)$.

33.3-4

Пусть $n > 1$ *и* $a \in \mathbb{Z}_n^*$. *Докажите, что функция* $f_a : \mathbb{Z}_n^* \to \mathbb{Z}_n^*$, *определяемая равенством* $f_a(x) = ax \bmod n$ *является перестановкой множества* $\mathbb{Z}_n^*$.

33.3-5

Выпишите все подгруппы групп $\mathbb{Z}_9$ *и* $\mathbb{Z}_{13}^*$.

33.4 Решение линейных диофантовых уравнений

Нас будут интересовать целочисленные решения уравнения

$$ax \equiv b \pmod{n}, \tag{33.22}$$

(здесь a, b *и* n *— целые числа; такие уравнения называют "линейными диофантовыми уравнениями"). Ясно, что здесь ва- жен лишь остаток от деления* x *на* n*, так что решением (33.22) естественно называть не целое число, а элемент группы* $\mathbb{Z}_n$ *(класс чисел, дающих один и тот же остаток при делении на* n*). Таким образом, можно сформулировать задачу так: есть элементы* $a, b \in \mathbb{Z}_n$*, мы ищем все* $x \in \mathbb{Z}_n$*, для которых* $ax = b \pmod{n}$.

Напомним, что через $\langle a \rangle$ *обозначается порождённая элементов* a *подгруппа (в данном случае подгруппа группы* $\mathbb{Z}_n$*). По определению* $\langle a \rangle = \{a^{(x)} : x > 0\} = \{ax \bmod n : x > 0\}$*, поэтому уравнение (33.22) имеет хотя бы одно решение тогда и только тогда, когда* $b \in \langle a \rangle$*. Сколько элементов в* $|\langle a \rangle|$*? По теореме Лагранжа (33.15) это число является делителем* n.

Теорема 33.20

Для любых положительных целых a *и* n

$$\langle a \rangle = \langle d \rangle = \{0, d, 2d, \ldots, ((n/d) - 1)d\}, \qquad (33.23)$$

и

$$|\langle a \rangle| = n/d,$$

где $d = gcd(a, n)$.

Доказательство

Расширенный алгоритм Евклида в применении к a *и* n *даёт тройку* (d, x', y')*, для которой* $d = gcd(a, n)$ *и* $ax' + ny' = d$*. Тогда* $ax' \equiv d \pmod{n}$*, и потому* $d \in \langle a \rangle$*. С другой стороны,* d *есть делитель* a*, и потому* $a \in \langle d \rangle$*. Следовательно,* $\langle a \rangle = \langle d \rangle = \{0, d, 2d, \ldots, ((n/d) - 1)d\}$

Следствие 33.21

Уравнение $ax \equiv b \pmod{n}$ *разрешимо относительно* x *тогда и только тогда, когда* $(a, n) \mid b$.

Следствие 33.22

Уравнение $ax \equiv b \pmod{n}$ *имеет* $d = gcd(a, n)$ *различных решений в* $\mathbf{Z}_n$ *или не имеет их вовсе.*

Доказательство Если уравнение $ax \equiv b \pmod{n}$ *имеет решение, то* $b \in \langle a \rangle$*. Согласно следствию 33.18, последовательность* $ax \bmod n$ *(*a *и* n *фиксированы,* $x = 0, 1, \ldots$*) периодична с периодом* $|\langle a \rangle| = n/d$*. Если* $b \in \langle a \rangle$*, то* b *встречается ровно один раз среди первых* n/d *членов рассматриваемой последовательности. При изменении* i *от 0 до* $n - 1$ *этот набор из* n/d *чисел проходится* d *раз и элемент* b *встречается* d *раз; соответствующие значения* x *служат решениями уравнения* $ax \equiv b \pmod{n}$.

Теорема 33.23

Пусть $d = gcd(a, n) = ax' + ny'$*, где* x' *и* y' *— целые числа (например, выдаваемые процедурой* Extended-Euclid*). Если* $d \mid b$*, то число* $x_0 = x'(b/d) \pmod{n}$ *является решением уравнения* $ax = b \pmod{n}$.

Доказательство

По условию $ax' \equiv d \pmod{n}$*, поэтому* $ax_0 \equiv ax'(b/d) \equiv d(b/d) \equiv b \pmod{n}$.

Теорема 33.24

Пусть уравнение $ax \equiv b \pmod{n}$ *разрешимо, и* x_0 *является его решением. Тогда уравнение имеет* $d = gcd(a,b)$ *решений в* $\mathbb{Z}_n$*, задаваемых формулой* $x_i = x_0 + i(n/d)$*, где* $i = 0, 1, 2, \ldots, n-1$.

Доказательство

Начав с x_0 *и двигаясь с шагом* n/d*, мы сделаем* d *шагов, прежде чем замкнём круг. При этом все эти числа будут оставаться решениями уравнения* $ax \equiv b \pmod{n}$*, так как при увеличении* x *на* n/d *произведение* ax *увеличивается на* $n(a/d)$*, то есть на кратное* n*. Таким образом, мы перечислили все* d *решений.*

В соответствии со сказанным напишем процедуру, которая по целым числам a*,* b *и* $n > 0$ *даёт все решения уравнения* $ax \equiv b \pmod{n}$.

```
Modular-Linear-Equation-Solver(a,b,n)
1  (d,x',y')\gets Extended-Euclid(a,n)
2  if d| b
3     then x_0 \gets x' (b/d) \bmod n
4          for i \gets 0 to d-1
5            do print (x_0+i(n/d)) \bmod n
6     else print "нет решений"
```

Например, для уравнения $14x \equiv 30 \pmod{100}$ *(*$a = 14$*,* $b = 30$ *и* $n = 100$*) вызов процедуры* Extended-Euclid *в строке 1 даёт* $(d, x, y) = (2, -7, 1)$*. Поскольку* $2 \mid 30$*, в строке 3 вычисляется* $x_0 = (-7) \cdot (15) \bmod 100 = 95$*, и в строках 4–5 печатаются числа* 95 *и* 45.

Процедура Modular-Linear-Equation-Solver(a, n) *выполняет* $O(\lg n + (a, n))$ *арифметических операций (*$O(\lg n)$ *в строке 1 и* $O(gcd(a, n))$ *в остальных строках).*

Следствие 33.25

Пусть $n > 1$*. Если* $gcd(a, n) = 1$*, то уравнение* $ax \equiv b \pmod{n}$ *имеет единственное решение (в* $\mathbb{Z}_n$*).*

Случай $b = 1$ *особенно важен — при этом мы находит* обратный к x элемент по модулю n *(multiplicative inverse modulo n), то есть обратный в группе* $\mathbb{Z}_n^*$ *элемент.*

Следствие 33.26

Пусть $n > 1$*. Если* $gcd(a, n) = 1$*, то уравнение*

$$ax \equiv 1 \pmod{n} \qquad (33.24)$$

имеет единственное решение в $\mathbb{Z}_n$*. При* $gcd(a, n) > 1$ *это уравнение решений не имеет.*

Тем самым мы научились вычислять обратный элемент в группе $\mathbb{Z}_n^*$ *за* $O(\lg n)$ *арифметических операций.*

Упражнения

33.4-1

Решите уравнение $35x \equiv 10 \pmod{50}$.

33.4-2

Докажите, что если $gcd(a, n) = 1$, *то из* $ax = ay \pmod{n}$ *следует уравнение* $x = y \pmod{n}$. *Покажите на пример, что условие* $gcd(a, n) = 1$ *существенно.*

33.4-3

Будет ли работать процедура MODULAR-LINEAR-EQUATION-SOLVER, *если строку 3 в ней заменить на*

```
3    then x_0 \gets x' (b/d) \bmod (n/d)
```

Объясните свой ответ.

*33.4-4**

Пусть $f(x) \equiv f_0 + f_1 x + \cdots + f_t x_t \pmod{p}$ *— многочлен степени* t *с коэффициентами* $f_i \in \mathbb{Z}_p$ *(где* p *— простое число). Назовём* $a \in \mathbb{Z}_p$ нулём *(zero) многочлена* f, *если* $f(a) \equiv 0 \pmod{p}$. *Докажите, что если* a *— нуль многочлена* f, *то найдётся многочлен* $g(x)$ *степени* $t-1$, *для которого* $f(x) \equiv (x-a)g(x) \pmod{p}$. *Выведите отсюда, что многочлен степени* t *имеет не более* t *нулей (в* $\mathbb{Z}_p$*).*

33.5 Китайская теорема об остатках

Около 100 г. до Р.Х. китайский математик Сун Цу (Sun-Tsŭ) решил такую задачу: найти число, дающее при делении на 3, 5 и 7 остатки 2, 3 и 2 соответственно (общий вид решения — $23 + 105k$ *при целых* k*). Поэтому утверждение об эквивалентности системы сравнений по взаимно простым модулям и сравнения по модулю произведений называют "китайской теоремой об остатках".*

Пусть некоторое число n *представлено в виде произведения попарно взаимно простых чисел* $n_1 n_2 \cdot \ldots \cdot n_k$. *Китайская теорема об остатках утверждает, что кольцо вычетов* $\mathbb{Z}_n$ *устроено как произведение колец вычетов* $\mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2} \times \cdots \times \mathbb{Z}_{n_k}$ *(с покомпонентным сложением и умножением). Это соответствие полезно и с алгоритмической точки зрения, так как бывает проще выполнить операции во всех множествах* $\mathbb{Z}_{n_i}$, *чем непосредственно в* $\mathbb{Z}_n$.

Теорема 33.27 (Китайская теорема об остатках)

Пусть $n = n_1 n_2 \cdots n_k$, *где* $n_1, n_2, \ldots, n_k$ *попарно взаимно просты. Рассмотрим соответствие*

$$a \leftrightarrow (a_1, a_2, \ldots, a_k), \tag{33.25}$$

где $a \in \mathbf{Z}_n$, $a_i \in \mathbf{Z}_{n_i}$ *и* $a_i \equiv a \bmod n_i$ *при* $i = 1, 2, \ldots, k$. *Формула (33.25) определяет взаимно однозначное соответствие между* $\mathbb{Z}_n$

и декартовым произведением $\mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2} \times \dots \times \mathbb{Z}_{n_k}$. При этом операциям сложения, вычитания и умножения в $\mathbb{Z}_n$ соответствуют покомпонентные операции над k-элементными кортежами: если

$$a \leftrightarrow (a_1, a_2, \dots, a_k)$$

и

$$b \leftrightarrow (b_1, b_2, \dots, b_k),$$

то

$$(a{+}b) \bmod n \leftrightarrow ((a_1{+}b_1) \bmod n_1, (a_2{+}b_2) \bmod n_2, \dots, (a_n{+}b_n) \bmod n_k), \tag{33.26}$$

$$(a{-}b) \bmod n \leftrightarrow ((a_1{-}b_1) \bmod n_1, (a_2{-}b_2) \bmod n_2, \dots, (a_n{-}b_n) \bmod n_k), \tag{33.27}$$

$$(ab) \bmod n \leftrightarrow ((a_1b_1) \bmod n_1, (a_2b_2) \bmod n_2, \dots, (a_nb_n) \bmod n_k). \tag{33.28}$$

Доказательство

Заметим прежде всего, что формула (33.25) действительно задаёт корректно определённое отображение $\mathbb{Z}_n$ в указанное произведение: если два числа сравнимы по модулю n, то их разность кратна n, и потому эти числа дают одинаковые остатки при делении на любое из n_i (так как $n_i \mid n$).

Обратное отображение также легко описать. Положим $m_i = n/n_i$, где $i = 1, 2, \dots, k$, то есть $m_i = n_1 n_2 \dots n_{i-1} n_{i+1} \dots n_k$. Очевидно, $m_i \equiv 0 \pmod{n_j}$ при $i \neq j$. Положим

$$c_i = m_i(m_i^{-1} \bmod n_i) \tag{33.29}$$

при $i = 1, 2, \dots, k$. Тогда $c_i \equiv 1 \pmod{n}_i$ и $c_i \equiv 0 \pmod{n}_j$ при $j \neq i$, и числу c_i соответствует набор с одной единицей на i-м месте:

$$c_i \leftrightarrow (0, 0, \dots, 0, 1, 0, \dots, 0).$$

Тем самым. положив

$$a \equiv (a_1c_1 + a_2c_2 + \dots a_kc_k) \pmod{n}. \tag{33.30}$$

мы получим число, соответствующее набору $(a_1, a_2, \dots, a_k)$. Тем самым для каждого набора можно найти соответствующий элемент $\mathbb{Z}_n$. Осталось убедиться, что такой элемент только один. Можно сослаться на то, что и слева, и справа у нас имеются множества из $n = n_1 n_2 \dots n_k$ элементов, и потому всякая сюръекция является биекцией. А можно заметить, что если a и a' дают одинаковые остатки при делении на все n_i, то $a - a'$ делится на все n_i и в силу взаимной простоты на их произведение, то есть на n (это легко следует из однозначности разложения на множители).

Рисунок 33.3 33.3 Китайская теорема об остатка: $n_1 = 5$, $n_2 = 13$. В столбцах перечислены различные остатки при делении на 13, в строках - при делении на 5. Каждое число от 0 до $65 - 1$ помещено в соответствующую строку и столбец, и теорема гарантирует, что в каждой клетке таблицы будет по одному числу.

Следствие 33.28

Если $n_1, n_2, \ldots, n_k$ попарно взаимно просты и $n = n_1 n_2 \cdots n_k$, то система сравнений

$$x \equiv a_i \pmod{n_i}$$

относительно x (где $i = 1, 2, \ldots, k$) имеет единственное решение по модулю n.

Следствие 33.29

Если $n_1, n_2, \ldots, n_k$ попарно взаимно просты, $n = n_1 n_2 \cdots n_k$, а x и a — целые числа, то свойство

$$x \equiv a \pmod{n}$$

равносильно выполнению сравнений

$$x \equiv a \pmod{n_i}$$

при всех $i = 1, 2, \ldots, k$.

В качестве примера рассмотрим систему сравнений

$$\begin{aligned} a &\equiv 2 \pmod{5}, \\ a &\equiv 3 \pmod{13}. \end{aligned}$$

Здесь $a_1 = 2$, $a_3 = 3$, $n_1 = m_2 = 5$, $n_2 = m_1 = 13$, и следовательно, $n = 65$. Поскольку $13^{-1} \equiv 2 \pmod{5}$ и $5^{-1} = 8 \pmod{13}$, мы находим

$$\begin{aligned} c_1 &\equiv 13(2 \bmod 5) \quad 26, \\ c_2 &\equiv 5(8 \bmod 13) \quad 40, \end{aligned}$$

и

$$\begin{aligned} a &\equiv 2 \cdot 26 + 3 \cdot 40 \pmod{65} \\ &\equiv 52 + 120 \pmod{65} \\ &\equiv 42 \pmod{65}. \end{aligned}$$

См. также рис. 33.3.

Таким образом, вычисления по модулю произведения взаимно простых чисел можно выполнять отдельно по модулю каждого из этих чисел.

Упражнения

33.5-1

Найдите все x, при которых $x \equiv 4 \pmod{5}$ и $x \equiv 5 \pmod{11}$.

33.5-2

Найдите все целые числа x, дающие при делении на 2, 3, 4, 5, 6 остатки 1, 2, 3, 4, 5 (соответственно).

33.5-3

Докажите, что (в контексте теоремы 33.27) при $gcd(a, n) = 1$ имеет место соответствие

$$(a^{-1} \bmod n) \leftrightarrow ((a_1^{-1} \bmod n_1), (a_2^{-1} \bmod n_2), \dots, (a_k^{-1} \bmod n_k)).$$

33.5-4

Пусть $f(x)$ — многочлен с целыми коэффициентами. Докажите, что (в условиях теоремы 33.27) число его корней по модулю n (вычетов, для которых $f(x) \equiv 0 \pmod{n}$) равно произведению чисел его корней по модулям $n_1, n_2, \dots, n_k$.

33.6 Степень элемента

Рассмотрим в мультипликативной группе вычетов $\mathbb{Z}_n^*$ последовательность степеней некоторого элемента a:

$$a^0, a^1, a^2, a^3, \dots . \qquad (33.31)$$

Мы начинаем счёт с нуля, полагая $a^0 \bmod n = 1$; i-й член последовательности равен $a^i \bmod n$. Например, последовательность степеней числа 3 по модулю 7 имеет вид

i	0	1	2	3	4	5	6	7	8	9	10	11	...
$3^i \bmod 7$	1	3	2	6	4	5	1	3	2	6	4	5	...

а для степеней числа 2 по модулю 7 имеем

i	0	1	2	3	4	5	6	7	8	9	10	11	...
$2^i \bmod 7$	1	2	4	1	2	4	1	2	4	1	2	4	...

В этом разделе под $\langle a \rangle$ мы будем понимать подгруппу группы $\mathbb{Z}_n^*$, порождённую элементом a, а под $ord_n(a)$ — порядок элемента a группы $\mathbb{Z}_n^*$. К примеру, $\langle 2 \rangle = \{1, 2, 4\}$ в группе $\mathbb{Z}_7^*$ и $ord_7(2) = 3$. Применив к группе $\mathbb{Z}_n^*$ следствие 33.19 и вспомнив определение φ–функции Эйлера, получим такое утверждение:

Теорема 33.30 (теорема Эйлера)

Если $n > 1$ — целое число, то

$$a^{\varphi(n)} \equiv 1 \pmod{n}. \qquad (33.22)$$

для всякого $a \in \mathbb{Z}_n^*$

При простом n эта теорема превращается в "малую теорему Ферма":

Теорема 33.31 (малая теорема Ферма)

Если $p > 1$ — простое число, то

$$a^{p-1} \equiv 1 \pmod p \qquad (33.23)$$

для всякого $a \in \mathbb{Z}_p^*$

Доказательство

Поскольку число p — простое, $\varphi(p) = p - 1$ (33.21).

Малая теорема Ферма применима ко всем элементам $\mathbb{Z}_p$, кроме нуля. Если умножить это сравнение на a. то получится сравнение $a^p \equiv a \pmod p$, которое верно и для $a = 0$ (p — любое простое число).

Если $\operatorname{ord}_n(g) = |\mathbb{Z}_n^*|$, *то все элементы $\mathbb{Z}_n^*$ являются степенями элемента g. В этом случае g называется* примитивным корнем *(primitive root) или* образующей *(generator) группы $\mathbb{Z}_n^*$. Например, 3 является примитивным корнем в $\mathbb{Z}_7^*$. Если группа $\mathbb{Z}_n^*$ имеет образующей, её называют* циклической *(cyclic). Имеет место следующая теорема (доказательство можно найти, например, в книге Нивена и Цукермана [151]).*

Теорема 33.32

Пусть $n > 1$. Группа $\mathbb{Z}_n^*$ циклична тогда и только тогда, когда n равно 2, 4, имеет вид p^k или $2p^k$ (где $p > 2$ — простое число, а k — целое положительное число).

Пусть g является образующей группы $\mathbb{Z}_n^*$. Тогда для всякого $a \in \mathbb{Z}_n^*$ найдётся z, для которого $g^z = a \pmod n$. Такое z называют дискретным логарифмом *(discrete logarithm) или* индексом *(index) элемента $a \in \mathbb{Z}_n^*$ по основанию g и обозначают* $\operatorname{ind}_{n,g}(a)$.

Теорема 33.33 (о дискретном логарифме)

Пусть g является образующей группы $\mathbb{Z}_n^*$. Тогда сравнение $g^x \equiv g^y \pmod n$ равносильно сравнению $x \equiv y \pmod{\varphi(n)}$.

Доказательство

Если $x \equiv y \pmod{\varphi(n)}$, то есть $x = y + k\varphi(n)$ при некотором целом k, то $g^x \equiv g^{y+k\varphi(n)} \equiv g^y \cdot (g^{\varphi}(n))^k \equiv g^y \cdot 1^k \equiv g^y$ (все равенства по модулю n).

Напротив, пусть $g^x \equiv g^y \pmod n$. Согласно следствию 33.18, последовательность степеней g периодична с периодом $|\langle g \rangle| = \varphi(n)$, поэтому что $x \equiv y \pmod{\varphi(n)}$.

Мы видим, что величина $ind_{n,g}(a)$ определена с точностью до слагаемого, кратного $\varphi(n)$, то есть может рассматриваться как элемент аддитивной группы $\mathbb{Z}_{\varphi(n)}^*$.

Вот пример использования понятия индекса:

Теорема При простом $p > 2$ и положительном целом k уравнение

$$x^2 \equiv 1 \pmod{p^k} \qquad (33.34)$$

имеет ровно два решения ($x = \pm 1$).

Доказательство

Если $x^2 \equiv 1 \pmod{p^k}$, *то* x *взаимно просто с* p, *поэтому решения надо искать в группе* $\mathbb{Z}_n^*$ *(где* $n = p^k$*); пусть* g *— образующая этой группы. Тогда уравнение (33.34) записывается в виде*

$$(g^{ind_{n,g}(x)})^2 \equiv g^{ind_{n,g}(1)} \pmod{n}. \tag{33.35}$$

По теореме 33.33 это можно переписать как

$$2 \cdot ind_{n,g}(x) \equiv 0 \pmod{\varphi(n)}. \tag{33.36}$$

(заметим, что индекс 1 *равен* 0*). Такие уравнения мы уже решали в разделе 33.4; теорема 33.24 говорит, что (33.36) имеет ровно два решения (а мы знаем, что* $x = \pm 1$ *подходит).*

[Впрочем, можно было бы обойтись и без ссылки на теорему 33.32: если $x^2 - 1 = (x-1)(x+1)$ *делится на* p^k, *то одна из скобок делится на* p, *тогда другая не делится, а значит, первая делится и на* p^k.*]*

Решение уравнения $x^2 \equiv 1 \pmod{n}$, *не сравнимое по модулю* n *с* ± 1, *называется* нетривиальным квадратным корнем из 1 в $\mathbb{Z}_n$ *(nontrivial square root of* 1 *modulo* n*). Например, 6 является нетривиальным корнем из 1 по модулю 35. Следующее утверждение понадобится нам в разделе 33.8 при обсуждении теста Миллера — Рабина.*

Следствие 33.35

Если $\mathbb{Z}_n$ *содержит нетривиальный корень из* 1, *то число* n *составное.*

Доказательство

Очевидно вытекает из теоремы 33.8.

Вычисление степеней повторным возведением в квадрат.

Возведение в степень по модулю играет важную роль при проверке чисел на простоту, а также в криптосистеме RSA. Как и для обычных чисел, повторное умножение — не самый быстрый способ; лучше воспользоваться алгоритмом повторного возведения в квадрат *(repeated squaring).*

Пусть мы хотим вычислить $a^b \bmod n$, *где* a *— вычет по модулю* n, *а* b *— целое неотрицательное число, имеющее в двоичной записи вид* $\langle b_k, b_{k-1}, \ldots, b_1, b_0 \rangle$ *(число знаков считаем равным* $k+1$*; старшие разряды, как обычно, слева). Мы вычисляем* $a^c \bmod n$ *для некоторого* c, *которое возрастает и в конце концов становится равным* b.

```
Modular-Exponentiation (a,b,n)
c \gets 0
d \gets 2
пусть \langle b_k,b_{k-1},\ldots,b_0\rangle ---
      двоичная запись $b$
for i \gets k downto 0
```

i	9	8	7	6	5	4	3	2	1	0
b_i	1	0	0	0	1	1	0	0	0	0
c	1	2	4	8	17	35	70	140	280	560
d	7	49	157	526	160	241	298	166	67	1

Рисунок 33.4 Работа процедуры MODULAR-EXPONENTATION при $a = 7$, $b = 560 = \langle 1000110000 \rangle$ и $n = 561$. Показаны значения переменных после очередного исполнения тела цикла **for**. Процедура возвращает ответ 1.

```
   do c \gets 2c
      d \gets (d \cdot d) \bmod n
      if b_i=1
        then c \gets c+1
             d \gets (d\cdot a) \bmod n
return d
```

При умножении c на 2 число a^c возводится в квадрат, при увеличении c на 1 число a^c умножается на a. На каждом шаге двоичная запись c сдвигается на 1 влево, после чего, если надо ($b_i = 1$), последняя цифра двоичной записи меняется с 0 на 1. (Заметим, что переменная c фактически не используется и может быть опущена.)

Оценим время работы процедуры. Если три числа, являющиеся её исходными данными, имеют не более β битов, то число арифметических операций есть $O(\beta)$, а число битовых — $O(\beta^3)$.

Пример ($a = 7, b = 560, n = 561$) показан на рис. 33.4.

Упражнения

33.6-1

Найдите порядки всех элементов в $\mathbb{Z}_{11}^*$. Найдите наименьшую образующую этой группы и укажите индексы всех элементов этой группы.

33.7-2

Предложите алгоритм вычисления $a^b \bmod n$, который обрабатывает биты двоичной записи b справа налево (от старших к младшим).

33.7-3

Объясните, как вычислить $a^{-1} \bmod n$ для $a \in \mathbb{Z}_n^*$ при помощи процедуры MODULAR-EXPONENTATION, *если известно значение $\varphi(n)$.*

33.7 Криптосистема RSA с открытым ключом

Криптосистемы с открытым ключом позволяют обмениваться секретными сообщениями по открытому каналу, не договариваясь заранее о ключе шифре; даже перехватив весь раз-

говор от начала до конца, враг не узнает секретного сообщения. Кроме того, эти же методы позволяют добавлять к сообщению "цифровую подпись", удостоверяющую, что сообщение не фальсифицировано врагами. Проверить аутентичность подписи легко, а а подделать её крайне трудно. Понятно, что такие методы находят широкое применение в банках, при подписывании контрактов, денежных переводах и т.п.

Криптосистема RSA основана на таком обстоятельстве: в настоящее время известны эффективные алгоритмы поиска больших простых чисел, но не известно сколько-нибудь приемлемого по времени работы алгоритма разложения произведения двух больших простых чисел на множители. Мы рассмотрим эти задачи (проверка простоты и разложение на множители) в разделах 33.8 и 33.9.

Криптосистемы с открытым ключом.

При использовании таких систем каждый участник переговоров имеет открытый ключ *(public key) и* секретный ключ *(secret key). В системе RSA ключ состоит из пары целых чисел. Участников переговоров может быть несколько. но для примера мы будем говорить о переговорах Алисы (A) и Боба (B). Их открытые ключи мы будем обозначать* P_A *и* P_B*, а секретные —* S_B *и* S_B.

Каждый участник сам создаёт два своих ключа. Секретный ключ он хранит в тайне, а открытый сообщает остальным участникам (и вообще всем желающим, например, через газеты или Internet; открытые ключи всех заинтересованных лиц можно публиковать в специальных справочниках и т.п.).

Обозначим через $\mathcal{D}$ *множество всех возможных сообщений (например, это может быть множество всех битовых строк). Потребуем, чтобы каждый ключ задавал перестановку множества* $\mathcal{D}$*, и через* $P_A()$ *и* $S_A()$ *будем обозначать перестановки, соответствующие ключам Алисы. Мы считаем, что каждая из перестановок* $P_A()$ *и* $S_A()$ *может быть быстро вычислена, если только известен соответствующий ключ.*

Мы хотим, чтобы ключи одного участника задавали взаимно обратные перестановки, то есть чтобы

$$M = S_A(P_A(M)), \tag{33.37}$$

и

$$M = P_A(S_A(M)). \tag{33.38}$$

было выполнено для любого сообщения $M \in \mathcal{D}$.

Самое главное — чтобы никто, кроме Алисы, не мог вычислять функцию $S_A()$ *за разумное время; именно на этом основаны все полезные свойства криптосистемы, перечисленные выше. Потому-то Алиса и держит значение* S_A *в секрете: если кто-либо узнает её секретный ключ, он сможет расшифровывать адресованные ей*

надписи:
encrypt - шифрование
communication channel - линия связи
eavesdropper - злоумышленник
decrypt - расшифровка
Боб, Алиса

Рисунок 33.5 33.5 Шифрование с открытым ключом. Боб шифрует сообщение M с помощью функции P_A и получает шифровку $C = P_A(M)$. Функции $S_A()$ и $P_A()$ взаимно обратны, поэтому Алиса может восстановить исходное сообщение M по шифровке: $M = S_A(C)$. Никто, кроме Алисы, не знает способа вычисления $S_A()$, поэтому сообщение M останется секретным, даже если злоумышленник подслушает C и знает P_A.

сообщения, подделывать её подпись или перевирать сообщения, которые она отправляет от своего имени. Главная трудность при разработке криптосистем состоит в том, чтобы придумать функцию $S_A()$*, для которой трудно было бы найти быстрый способ вычисления, даже зная такой способ для обратной функции* $P_A()$.

Опишем процесс пересылки шифрованного сообщения. Допустим, Боб желает послать Алисе секретное сообщение. Это происходит так:

- *Боб находит* P_A *— открытый код Алисы (по справочнику или прямо от Алисы)*
- *Боб зашифровывает своё сообщение* M *и посылает Алисе* шифровку *(ciphertext)* $C = P_A(M)$.
- *Алиса получает* C *и восстанавливает изначальное сообщение* $M = S_A(C)$.

Этот процесс показан на рис. 33.5.

Теперь объясним, как снабдить сообщение электронной подписью. Пусть Алиса хочет послать Бобу ответ M'*, подписанный электронной подписью (рис. 33.6).*

- *Алиса вычисляет* электронную подпись *(digital signature)* $\sigma = S_A(M')$.
- *Алиса посылает Бобу пару* (M', σ)*, состоящую из сообщения и подписи.*
- *Боб получает пару* (M', σ) *и убеждается в подлинности подписи, проверив равенство* $M' = P_A(\sigma)$.

Если участников переговоров много, будем считать, что каждое сообщение M' *должно начинаться с имени отправителя — прочтя его, можно узнать, чей ключ надо использовать для проверки. Если равенство выполняется, то можно быть уверенным в том, что сообщение действительно было послано отправителем и дошло в неизменённом виде. Если же равенство не выполнено, сообщение было повреждено помехами или фальсифицировано.*

sign - вычисление подписи
verify - проверка подписи
accept - сообщение подлинное

Рисунок 33.6 33.6 Цифровая подпись в системе с открытым ключом. Алиса подписывает своё сообщение M', прикладывая к нему цифровую подпись $\sigma = S_A(M')$. Боб, получая от Алисы пару (M', σ), проверяет соотношение $M' = P_A(\sigma)$. Если оно выполняется, подпись и само сообщение подлинны.

Таким образом, цифровая подпись выполняет функции обычной. Подлинность цифровой подписи может проверить каждый, знающий открытый ключ Алисы. Прочитав сообщение Алисы, Боб может переслать другим участникам переговоров, и те тоже смогут убедиться в его подлинности. Например, подписанным документом может быть банковское поручение о перечислении денег со счёта Алисы на счёт Боба — и банк будет знать, что оно настоящее, а не фальсифицировано Бобом.

При таком сценарии содержимое подписанного сообщения не является секретным. Можно добиться и этого, скомбинировав два описанных приёма. Отправитель, желающий зашифровать и подписать своё сообщение, должен сначала приложить к своему сообщению цифровую подпись, а затем зашифровать пару (сообщение, подпись) при помощи открытого ключа получателя. Получатель сначала расшифрует эту пару с помощью своего секретного ключа, а затем проверит её подлинность с помощью открытого ключа отправителя. Аналогичная процедура с бумажным документом могла бы выглядеть так: письмо пишется от руки, подписывается и вкладывается в конверт, который может открыть только получатель.

Криптосистема RSA.

Чтобы построить пару ключей для криптосистемы RSA *(RSA cryptosystem), надо сделать следующее:*

1. *Взять два больших простых числа p и q (скажем, около* 100 *десятичных цифр в каждом).*
2. *Вычислить $n = pq$.*
3. *Взять небольшое нечётное число e, взаимно простое с $\varphi(n)$. (Из соотношения (33.20) следует, что $\varphi(n) = (p-1)(q-1)$.)*
4. *Вычислить $d = e^{-1} \bmod \varphi(n)$. (По Следствию 33.26 d существует и определено, по модулю $\varphi(n)$ однозначно.)*
5. *Составить пару $P = (e, n)$* — открытый RSA-ключ *(RSA public key).*
6. *Составить пару $S = (d, n)$* — секретный RSA-ключ *(RSA secret key).*

Множеством $\mathcal{D}$ всех возможных сообщений для этой криптосистемы является $\mathbb{Z}_n$. Открытому ключу $P = (e, n)$ соответ-

ствует преобразование

$$P(M) = M^e \bmod n, \tag{33.39}$$

а секретному ключу $S = (d, n)$ *— преобразование*

$$S(C) = C^d \bmod n. \tag{33.40}$$

Как уже говорилось, эти преобразования можно использовать и для шифрования, и для электронных подписей.

Для возведения в степень в формулах !(33.39)-(33.40) разумно пользоваться процедурой Modular-Exponentation *из Раздела 33.6. Если считать, что числа* d *и* n *имеют порядка* β *битов, а число* e *имеет* $O(1)$ *битов, то преобразование* P *потребует* $O(1)$ *умножений по модулю* n *(*$O(\beta^2)$ *битовых операций), а а преобразование* S $O(\beta)$ *умножений (*$O(\beta^3)$ *битовых операций) (разумеется, при известном ключе).*

Теорема 33.36 (Корректность системы RSA)

Формулы (33.39) и (33.40) задают взаимно обратные перестановки множества $\mathbb{Z}_n$.

Доказательство

Очевидно,

$$P(S(M)) = S(P(M)) = M^{ed} \pmod n$$

для всякого $M \in \mathbb{Z}_n$*. Мы знаем, что* e *и* d *взаимно обратны по модулю* $\varphi(n)$*, то есть*

$$ed = 1 + k(p-1)(q-1)$$

для некоторого целого k*. Если* $M \not\equiv 0 \pmod p$*, то по малой теореме Ферма (теорема 33.31) имеет*

$$M^{ed} \equiv M(M^{p-1})^{k(q-1)} \equiv M \cdot 1^{k(q-1)} \equiv M$$

по модулю p*. Равенство* $M^{ed} \equiv M \pmod p$*. выполнено, конечно, и при* $M \equiv 0 \pmod p$*, так что оно верно для всех* M*. По тем же причинам* $M^{ed} \equiv M \pmod q$*, и потому (следствие 33.29)* $M^{ed} \equiv M \pmod n$ *при всяком* M*.*

Надёжность криптосистемы RSA основывается на трудности задачи разложения составных чисел на множители: если враг разложит (открыто опубликованное) число n *на множители* p *и* q*, он сможет найти* d *тем же способом, что и создатель ключа. Таким образом, если задача разложения на множители может быть решена быстро (каким-то пока неизвестным нам алгоритмом), то " взломать" криптосистему RSA легко. Обратное утверждение, показывающее, что если задача разложения на* *множители сложна, то взломать систему RSA трудно, не доказано — однако за время существования этой системы никакого иного способа её сломать обнаружено не было.*

Как мы увидим в разделе 33.9, разложение чисел на множители — дело непростое, быстрого алгоритма для этого мы не знаем; известные ныне методы не позволяют разложить на множители произведение двух 100*-значных простых чисел за разумное время — для этого нужны какие-то новые идеи и методы (если это вообще возможно).*

Конечно, надёность системы RSA зависит от размера простых чисел, поскольку небольшие числа легко разложить на множители. Поэтому надо уметь искать большие простые числа. Этой задачей мы займёмся в разделе 33.8.

На практике (для ускорения вычислений криптосистему RSA часто используют вместе с какой-то традиционной системой шифрования, в которой ключ необходимо хранить в секрете. Выбрав такую систему, мы используем для шифрования её — а система RSA используется только для передачи секретного ключа, который может быть значительно короче самого сообщения. Сам этот ключ может выбираться, например, случайно и только для один раз.

Похожий подход применяется для ускорения работы с цифровыми подписями. Система RSA используется при этом в паре с так называемой односторонней хеш-функцией *(one-way hash function). Такая функция отображает каждое сообщение M в достаточно короткое сообщение $h(M)$ (например,* 128*-битовую строку), при этом $h(M)$ легко вычислить по M, но не удаётся найти два разных сообщения M и M', для которых $h(M) = h(M')$ (хотя таких пар много по принципу Дирихле).*

Образ $h(M)$ сообщения M можно сравнить с "отпечатком пальца" (fingerprint) сообщения M. Алиса, желая подписать своё сообщение M, вычисляет $h(M)$, который и шифрует своим секретным RSA-ключом. Затем она посылает Бобу пару $(M, S_A(h(M)))$. Боб удостоверяется в подлинности подписи, проверив, что $P_A(S_A(h(M))) = h(M)$. Конечно, можно фальсифицировать текст сообщения, найдя другое сообщение M', для которого $h(M) = h(M')$, но это (по нашему предположению) сложно.

*Конечно, при использовании открытых ключей надо ещё убедиться, что сами ключи не были подменены. Предположим, что имеется некоторый "нотариус", честность которого вне подозрений и открытый ключ которого все знают (и в его правильности не сомневаются). Нотариус может выдавать известным ему людям справки (*сертификаты*, certificates) о том, что их открытый ключ такой-то, подписывая эти справки собственной цифровой подписью. (Приходящие должны сообщить нотариусу свой* *открытый ключ.) Подлинность сертификата может быть проверена каждым, кому известен открытый ключ нотариуса; любой зарегистрированный у нотариуса участник переговоров может прилагать с своим сообщениям выданный нотариусом сертификат.*

Упражнения

33.7-1

Вычислите открытый и секретный RSA-ключи при $p = 11$, $q = 29$, $n = 319$ *и* $e = 3$. *Чему равно* d *в секретном ключе? Зашифруйте сообщение* $M = 100$.

33.7-2

Пусть значение e *для открытого кода равно* 3. *Докажите, что враг, узнавший значение* d, *сможет разложить* n *на множители за время, полиномиальное относительно длины двоичной записи* n. *(Аналогичное утверждение верно и для произвольного* e, *см. работу Миллера [147].)*

*33.7-3**

Докажите, что система RSA мультипликативна, то есть

$$P_A(M_1)P_A(M_2) \equiv P_A(M_1M_2) \pmod{n}$$

для любых $M_1, M_2 \in \mathbb{Z}_n$. *Предположим, что враг имеет способ быстро расшифровывать 1% всех сообщений из* $\mathbf{Z}_n$. *Используя мультипликативность системы RSA, объясните, как тогда он может с высокой вероятностью достаточно быстро расшифровать любое сообщение.*

33.8 Проверка чисел на простоту

В этом разделе мы обсуждаем вопрос о том, как искать большие простые числа.

Распределение простых чисел.

Как найти большое простое число? Естественный подход таков: взять большое случайное число и посмотреть, не окажется ли оно простым. Если нет, попробовать другое случайное число и так далее. Этот способ пригоден, лишь если простые числа не слишком редки — к счастью, это действительно так, как показывает теорема о распределении простых чисел, доказанная в конце прошлого века. Вот её формулировка (доказательство далеко выходит за рамки этой книги).

Определим функцию распределения простых чисел *(prime distribution function)* π, *положив* $\pi(n)$ *равным количеству простых чисел, не превосходящих* n. *Например, на отрезке от* 1 *до* 10 *есть* 4 *простых числа* 2, 3, 5 *и* 7, *поэтому* $\pi(10) = 4$.

Теорема 33.37 (асимптотический закон распределения простых *чисел)*

$$\lim_{n\to\infty}\frac{\pi(n)}{n/\ln n}=1.$$

Выражение $n/\ln n$ даёт неплохое приближение к $\pi(n)$ даже при небольших значениях n. Уже при $n = 10^9$ (небольшое число с точки зрения специалистов по теории чисел) погрешность приближения не превосходит 6% ($\pi(10^9) = 50\,847\,478$, $10^9/10^9\ln 10^9 \approx 48\,254\,942$).

Асимптотический закон позволяет оценить вероятность, с которой целое число, наугад выбранное из отрезка от 1 до n, является простым — это примерно $1/\ln n$. Тем самым для отыскания простого числа от 1 до n нужно проверить на простоту порядка $\ln n$ случайно выбранных чисел. Например, чтобы найти простое число из 100 десятичных знаков, надо перебрать пордяка $\ln 10^{100} \approx 230$ случайных чисел от 1 до 10^{100}. (Эта оценка по очевидным причинам может быть уменьшена вдвое: проверять на простоту стоит только нечётные числа. Заметим также, что число десятичных знаков в наугад взятом числе от 1 до 10^{100} с большой вероятностью близко к 100 — около 90% всех чисел в этом диапазоне имеют 100 знаков, около 99% — 99 или более знаков, около 99,9% — 98 или более знаков и т.д.)

Нам остаётся объяснить, каким образом можно проверить, будет ли простым данное большое число n. Другими словами, мы хотим узнать, состоит ли разложение числа n на простые множители

$$n = p_1^{e_1}p_2^{e_2}\cdots p_r^{e_r},$$

($r \geqslant 1$; числа $p_1, p_2, \dots, p_r$ — различные простые делители n) из единственного простого числа ($r = 1$, $e_1 = 1$).

Самый простой способ проверки — перебор делителей *(trial division). Будем пытаться разделить n на $2, 3, \dots, \lfloor\sqrt{n}\rfloor$ (чётные числа, большие 2, можно пропускать). Если n не делится ни на одно из этих чисел, то оно простое (если n разлагается в произведение двух или более множителей, то один из множителей не превосходит $\sqrt{n}$). Но дело это долгое — уже число делений есть $\Theta(\sqrt{n})$, и время работы экспоненциально зависит от длины записи числа n (напомним, что двоичное представление n занимает $\beta = \lfloor\lg(n+1)\rfloor$ битов, так что $\sqrt{n} = \Theta(2^{\beta/2})$). Таким образом, такой подход применим, лишь есть число n мало, или имеет небольшой делитель.*

Если при переборе делителей мы обнаруживаем, что число n составное, то одновременно находится и делитель числа n. Для других способов проверки простоты это не так — можно убедиться, что число составное, так и не указав никакого его делителя. Это не удивительно, так как задача разложения числа на множители, по-видимому, гораздо сложнее задачи проверки про-

стоты числа. К задаче разложения на множители мы вернёмся в следующем разделе.

Псевдопростые числа.

Сейчас мы опишем "почти правильный" алгоритм проверки числа на простоту, приемлемый для многих практических приложений. (Небольшое усложнение этого алгоритма, делающее его совсем правильным, будет описано дальше.)

Обозначим через $\mathbb{Z}_n^+$ множество всех ненулевых вычетов по модулю n:

$$\mathbf{Z}_n^+ = \{1, 2, \ldots, n-1\}.$$

Если число n простое, то $\mathbb{Z}_n^+ = \mathbb{Z}_n^*$.

Назовём число n псевдопростым по основанию a (base-a pseudoprime), если выполнено утверждение малой теоремы Ферма:

$$a^{n-1} \equiv 1 \pmod{n}. \tag{33.42}$$

Любое простое число n является псевдопростым по любому основанию $a \in \mathbb{Z}_n^+$. Поэтому если нам удалось найти основание a, по которому n не является псевдопростым, то мы можем быть уверены, что n — составное. Оказывается, что во многих случаях достаточно проверить основание $a = 2$

```
Pseudoprime(n)
if Modular-Exponentiation(2,n-1,n) \not\equiv 1 \pmod n
   then return composite     (заведомо)
   else return prime         (возможно)
```

Эта процедура может совершать ошибки, но только в одну сторону: если она сообщает, что число n составное, то это действительно так, но она может принять составное число за простое (если оно является псевдопростым по основанию 2). Как часто такое происходит? Оказывается, не так уж часто — среди чисел до 10 000 есть только 22 числа, для которых процедура Pseudoprime даёт неверный ответ (первые четыре из них — 341, 561, 645 и 1105). Можно показать, что доля таких "плохих" чисел среди β-значных чисел стремится к 0 при $\beta \to \infty$. Используя оценки из работы Померанца [157], можно показать, что доля составных чисел среди 50-разрядных псевдопростых по основанию 2 чисел не превосходит 10^{-6}, а среди 100-разрядных псевдопростых чисел — 10^{-13}.

Нет никаких причин ограничиваться проверкой лишь основания 2; можно проверять соотношение (33.42) и при других a. Но это не всегда помогает: существуют составные числа n, которые являются псевдопростыми по любому основанию $a \in \mathbb{Z}_n^*$. Такие числа называются числами Кармайкла (Carmichael numbers). Три первых числа Кармайкла таковы: 561, 1105 и 1729. Числа Кармайкла редки: среди первых ста миллионов положительных чисел их *имеется всего 255. (В упр. 33.8-2 мы укажем одну из причин, по которым таких чисел мало.)*

Сейчас мы исправим наш тест так, чтобы даже числа Кармайкла не смогли ввести его в заблуждение.

Вероятностный тест Миллера — Рабина

Этот вероятностный алгоритм проверки простоты числа получается из процедуры Pseudoprime *двумя принципиальными усовершенствованиями.*

- *Тест Миллера — Рабина проверяет соотношение (33.42) для нескольких значений a.*
- *Вычисляя степень a, фигурирующую в сравнении (33.42), тест по ходу вычисления проверят, не обнаружился ли нетривиальный корень из* 1 *по модулю n. Если да, тест немедленно прекращает работу и сообщает, что число n — составное (следствие 33.35).*

Процедура Miller-Rabin, *приведённая ниже, ниже, получает на вход нечётное число $n > 2$ (простоту которого мы хотим проверить) и и целое положительное число s, определяющее, сколько итераций надо провести (чем больше s, тем меньше вероятность ошибки). Тест использует генератор случайных чисел* Random *(см. раздел 8.3); процедура* Random$(1, n-1)$ *возвращает случайное целое число a, равномерно распределённое на отрезке $1 \leqslant a \leqslant n-1$.*

Тест использует процедуру Witness: Witness(a, n) *истинно, если a является ” свидетелем” того, что число n составное.*

```
Witness(a,n)
1 пусть \langle b_k,b_{k-1},\ldots,b_0$ --- двоичная запись $n-1$
2 d \gets 1
3 for i \gets k downto 0
4   do x \gets d
5      d \gets (d\cdot d) \bmod n
6      if d=1 и $x\ne 1$ и $x\ne n-1$
7         then return true
8      if b_i=1
9         then d \gets (d \cdot a) \bmod n
10 if d\ne 1
11   then return true
12 return false
```

Вот как происходит эта проверка. В строке 1 число $n-1$ переводится в двоичную систему. Это представление нужно, чтобы вычислять $a^{n-1} \bmod n$ методом повторного возведения в квадрат (строки 3–9). Это делается тем же способом, что и в процедуре Modular-Exponentation, *но заодно в строках 6–7 проверяется, не наткнулись ли мы на нетривиальный корень из* 1. *Если наткну-*

лись, то продолжать дальше вычисление $a^{n-1} \bmod n$ *нет смысла, мы и так знаем, что число* n *составное; процедура* Witness *возвращает значение* TRUE*. Если этого не произошло, возведение в степень заканчивается с результатом* d *и в строках 10–11 мы проверяем, равно ли* d *единице по модулю* n*. Если нет, то число* n *также составное (процедура возвращает значение* TRUE*). Если нет, то возвращается значение* FALSE *("число* a *не является свидетелем того, что* n *составное").*

В каком случае процедура Witness(a, n) *возвращает значение* TRUE*? В двух — либо имеется нетривиальный корень из* 1 *по модулю* n*, либо не выполнена малая теорема Ферма (число* n *не является псевдопростым по основанию* a*). В обоих случаях число* n *является составным.*

Приведём теперь полностью вероятностный алгоритм Миллера — Рабина.

```
1 for j \gets 1 to s
2   do a \gets Random(1,n-1)
3     if Witness(a,n)
4       then return составное      заведомо
5 return простое                   почти наверняка
```

Процедура Miller-Rabin *берёт* s *случайных чисел от* 1 *до* $n-1$ *и проверяет, не является ли какое-то из них свидетелем того, что* n *— составное (в описанном выше смысле). Если является, то число* n *заведомо является составным, и об этом можно сообщить в строке 4. Если ни одно из выбранных чисел не является свидетелем, то выдаётся ответ "простое", хотя гарантии тут нет. (Мы оценим далее, насколько вероятен такой ответ при составном* n*.)*

Для примера рассмотрим работу этого алгоритма для числа 561 *(которое, как мы говорили, является числом Кармайкла 561). Для этого вернёмся к рис. 33.4, который соответствует вычислениям для* $a = 7$*. При последнем возведении в квадрат процедура обнаруживает нетривиальный корень из* 1*, поскольку* $7^{280} \equiv 67 \pmod{561}$ *и* $7^{560} \equiv 1 \pmod{561}$*. Тем самым значение* $a = 7$ *является свидетелем того, что число* 561 *составное:* Witness$(7, 561) =$ TRUE*. Если* $a = 7$ *встретится среди* s *случайных значений, выбранных в алгоритме* Miller-Rabin*, то алгоритм выдаст ответ " составное".*

Если двоичная запись n *содержит* β *битов, то процедура* Miller-Rabin(s, n) *выполняет* $O(s\beta)$ *арифметических (и* $O(s\beta^3)$*) битовых) операций (производится не более* s *операций возведения в степень).*

Оценка вероятности ошибки для теста Миллера — Рабина.

Сообщая, что число n *является простым, процедура* Miller-Rabin *(как и* Pseudoprime*) может ошибаться. Но тут есть* *важная разница. Процедура* PSEUDOPRIME *давала неправильный ответ для некоторой (пусть небольшой) доли всех чисел — такие числа можно назвать "плохими" с точки зрения этой процедуры. Для теста Миллера — Рабина плохих чисел нет: для любого числа тест даёт правильный ответ с большой вероятностью. Плохим может быть лишь случайный выбор пробных свидетелей, и вероятность этого события мала.*

Как мы сейчас докажем, для любого составного числа n не менее половины всех возможных значений a позволят установить, что n составное — так что вероятность однократного неудачного выборпа свидетеля не превышает $1/2$, а при s-кратном повторении падает до 2^{-s}.

Теорема 33.38

Пусть n — нечётное составное число. Тогда среди элементов $\mathbb{Z}_n^+$ не менее $(n-1)/2$ являются свидетелями того, что n составное, то есть

$$\left|\{a \in \mathrm{Z}_n^+ : \textsc{Witness}(a,n) = \textsc{true}\right| \geqslant (n-1)/2.$$

Доказательство

Пусть фиксировано начётное составное число n. Будем называть элементы $a \in \mathbb{Z}_n^+$, для которых $\textsc{Witness}(a,n) = \textsc{false}$, *плохими. Мы хотим показать, что число плохих элементов не превосходит $(n-1)/2$.*

Заметим, что все негодные элементы лежат в $\mathbb{Z}_n^*$. В самом деле, для плохого a выполнено равенство $a^{n-1} \equiv 1 \pmod{n}$, и потому a^{n-1} и само a взаимно просты с n.

Теперь мы покажем, что все плохие элементы содержатся в некоторой собственной подгруппе B группы $\mathbb{Z}_n^*$. По следствию 33.16 число элементов в B не больше половины общего числа эдементов в $\mathbb{Z}_n^*$, откуда и следует доказываемое утверждение. Остаётся найти собственную подгруппу B, содержащую все плохие элементы.

Случай 1: Существует $x \in \mathbb{Z}_n^*$, для которого

$$x^{n-1} \not\equiv 1 \pmod{n}. \tag{33.43}$$

Тогда положим B равным $\{b \in \mathbb{Z}_n^* : b^{n-1} \equiv 1 \pmod{n}\}$. Множество B замкнуто относительно операции $\cdot_n$, поэтому по теореме 33.14 оно является подгруппой группы $\mathbb{Z}_n^*$. Очевидно, все плохие элементы лежат в B и по предположению группа B является собственной подругоой — так что в этом случае всё доказано.

Случай 2: Для всякого $x \in \mathbb{Z}_n^*$

$$x^{n-1} \equiv 1 \pmod{n} \tag{33.44}$$

(n является числом Кармайкла).

Этот случай несколько более сложен. Покажем прежде всего, что n не может быть степенью простого числа. Пусть это не так и $n = p^e$ для нечётного простого p и для целого $e > 1$. Тогда по теореме 33.32 группа $\mathbb{Z}_n^*$ является циклической, то есть содержит элемент g, для которого $ord_n(g) = |\mathbb{Z}_n^*| = \varphi(n) = (p-1)p^{e-1}$. По теореме 33.33 из сравнения (33.44) следует, что $n - 1 \equiv 0 \pmod{\varphi(n)}$, но это невозможно, так как правая часть делится на p (и даже на p^{e-1}, а левая на единицу меньше числа n, делящегося на p.

Итак, n является составным числом, которое не есть степень никакого простого числа, и потому содержит несколько разных простых множителей. Поэтому мы можем разложить n в произведение двух взаимно простых чисел $n_1, n_2 > 1$ (например, включив в n_1 один простой множитель, а в n_2 — все остальные).

Число $n-1$ чётно. Выделим из него наибольшую степень двойки 2^t, то есть запишем его в виде $n - 1 = 2^t u$, где u нечётно. Для каждого $a \in \mathbb{Z}_n^+$ рассмотрим последовательность

$$\hat{a} = \langle a^u, a^{2u}, a^{2^2u}, \ldots, a^{2^tu} \rangle. \tag{33.45}$$

состоящую из вычетов по модулю n; каждый следующий её элемент получается из предыдущего возведением в квадрат, а последний элемент есть a^{n-1}. Заметим, что двоичная запись числа $n-1$ оканчивается на n нулей. и потому все элементы этой последовательности встретятся в качестве промежуточных результатов при вычислениях в процедуре Witness *(которые заканчиваются t возведениями в квадрат).*

Какой вид может иметь последовательность $\hat{a}$ при различных a? Для каждого $j = 0, 1, \ldots, t$ посмотрим, какие члены могут появляться на j-ом месте в последовательности a (для удобства начинаем нумерацию с нуля, считая a^u нулевым членом последовательности $\hat{a}$). Последним членом ($j = t$) может быть только единица (иначе мы имели бы уже рассмотренный случай 1). С другой стороны, в начале последовательности $\hat{a}$ (то есть для $j = 0$) может стоять не только 1, но и -1 (при $a = -1$ имеем $a^u = -1$, так как u нечётно), а также, возможно, и другие числа.

Рассмотрим наибольшее значение $j \in \{0, 1, \ldots, t\}$, для которого существует такой элемент $v \in \mathbb{Z}_n^*$, что $v^{2^yu} \equiv -1 \pmod{n}$. (При $j = 0$, как мы видели, такой элемент v найдётся.) Пусть $v \in \mathbb{Z}_n^*$ — один из твких элементов, то есть $v^{2^ju} \equiv -1 \pmod{n}$. (Этот элемент пригодится нам чуть позже.)

Положим

$$B = \{x \in \mathbb{Z}_n^* : x^{2^ju} \equiv \pm 1 \pmod{n}\}.$$

Поскольку B замкнуто относительно умножения по модулю n, оно является подгруппой группы $\mathbb{Z}_n^*$.

Покажем, что любой плохой элемент a принадлежит B. Посмотрим на последовательность $\hat{a}$, порождённую плохим элементом, и двигаясь справа налево. На её последнем месте стоит единица. Перед единицей может стоять либо ещё одна единица, либо число -1 (иначе был бы обнаружен нетривиальный корень из единицы и элемент a не был бы плохим). При этом число -1 может появиться лишь в позиции j или левее (так как j было выбрано максимальным). Значит, в j-ой позиции находится либо 1, либо -1, то есть $a \in B$.

Теперь мы воспользуемся существованием элемента v, чтобы показать, что B является собственной подгруппой, построив $w \in \mathbb{Z}_n^* \setminus B$. По следствию 33.29 из $v^{2^j u} \equiv -1 \pmod{n}$ следует, что $v^{2^j u} \equiv -1 \pmod{n_1}$. По следствию 33.28 найдётся элемент w, для которого одновременно выполнены условия

$$\begin{aligned} w &\equiv v \pmod{n_1}, \\ w &\equiv 1 \pmod{n_2}. \end{aligned}$$

Тогда

$$\begin{aligned} w^{2^j u} &\equiv -1 \pmod{n_1}, \\ w^{2^j u} &\equiv 1 \pmod{n_2}, \end{aligned}$$

откуда видно (следствие 33.29), что

$$w^{2^j u} \not\equiv \pm 1 \pmod{n}, \tag{33.46}$$

Из двух последних сравнений по модулям n_1 и n_2 видно, что w взаимно просто с n_1 и с n_2 и тем самым взаимно просто с n. Следовательно, w принадлежит $\mathbb{Z}_n^* \setminus B$, что завершает доказательство теоремы — мы установили, что все плохие элементы принадлежат собственной подгруппе, и потому их число не превосходит $(n-1)/2$.

Теорема 33.39

Для нечётного $n > 2$ нечётно и для любого целого положительного s вероятность ошибки процедуры Miller-Rabin(n, s) *не превосходит 2^{-s}.*

Доказательство

Для простого n вообще не может быть ошибки. Если же n составное, то по теореме 33.38 при каждом случайном выборе a в строке 2 мы с вероятностью $1/2$ или более обнаружим свидетеля того, что n составное. Поэтому вероятность того, что этого не произойдёт при s (независимых) попытках, не превосходит 2^{-s}.

Взяв, например, $s = 50$, мы получим вероятность ошибки 2^{-50}. Такими малыми вероятностями на практике обычно пренебрегают. Если применять тест к случайно выбранному числу, то вероятность ошибки будет ещё меньше. Дело в том, что *наша оценка (свидетелей не меньше половины) для большинства n верна с большим запасом. Для большинства n количество плохих значений a обычно значительно меньше $(n-1)/2$, и на практике можно ограничиться, скажем, тремя попытками ($s = 3$) с достаточно малой вероятностью ошибиться. Однако если n не выбирается случайным образом, а даётся нам извне, то значительно улучшить оценку $(n-1)/2$ для числа плохих элементов нельзя (хотя немного можно: число плохих элементов, как можно доказать, не превосходит $(n-1)/4$ — и эта оценка уже неулучшаема).*

Упражнения

33.8-1

Докажите, что если целое число n не является простым и не является степенью простого числа, то в $\mathbb{Z}_n$ существует нетривиальный корень из 1.

*33.8-2**

Можно несколько усилить теорему Эйлера, показав, что

$$a^{\lambda(n)} \equiv 1 \pmod n,$$

для всех a, если $\lambda(n)$ определить формулой

$$\lambda(n) = lcm(\varphi(p_1^{e_1}), \ldots, \varphi(p_r^{e_r})). \qquad (33.47)$$

(при $n = p_1^{e_1} \ldots p_r^{e_r}$). Докажите, что $\lambda(n) \mid \varphi(n)$. Докажите, что составное число n является числом Кармайкла тогда и только тогда, когда $\lambda(n) \mid n-1$. Например, для числа $561 = 3 \cdot 11 \cdot 17$ (наименьшее число Кармайкла) им имеем $\lambda(n) = lcm(2, 10, 16) = 80 \mid 560$. Докажите, что всякое число Кармайкла "свободно от квадратов" (не делится на квадрат никакого простого числа) и имеет по меньшей мере 3 простых делителя. (Это одна из причин, почему числа Кармайкла редки.)

33.8-3

Пусть x является нетривиальным корнем из 1 по модулю n. Докажите, что $gcd(x-1, n)$ и $gcd(x+1, n)$ являются нетривиальными (отличными от 1 и от n) делителями числа n.

**33.9 Разложение чисел на множители.*

Пусть нам дано число n, которое надо разложить на множители *(to factor it), то есть представить в виде произведения простых чисел. Тест из предыдущего раздела позволяет установить, что n составное, но ничего не говорит об его множителях. Это не случайно — насколько мы можем судить сегодня, разложение на множители значительно сложнее проверки простоты. Даже наиболее мощные современные суперкомпьютеры, использующие наиболее совершенные (из известных на сегодняшний день) алгоритмы, не способны за разумное время разложить на множители наугад взятое 200-значное число.*

ρ—эвристика Полларда.

Перебирая все числа от 1 до B в качестве возможных делителей заданного числа n, мы сможем разложить на множители любое число вплоть до B^2. Сейчас мы опишем алгоритм, который делает примерно столько же действий и позволяет раскладывать на множители числа порядка B^4 (в большинстве случаев). К сожалению, нельзя гарантировать, что он выдаст ответ достаточно быстро; нельзя гарантировать даже того, что вообще разложение будет найдено. Однако на практике этот алгоритм зарекомендовал себя неплохо.

```
Pollard-Rho(n)
1 i \gets 1
2 x_i \gets Random (0,n-1)
3 y \gets x_i
4 k \gets 2
5 while true
6   do i \gets i+1
7      x_i \gets (x_{i-1}^2 - 1) \bmod n
8      d \gets \НОД(y-x_i,n)
9      if d \ne 1 и d \ne n
10       then print d
11     if i=k
12       then y \gets x_i
13            k \gets 2k
```

Эта процедува основана на рекуррентном соотношении

$$x_i = (x_{i-1}^2 - 1) \bmod n \tag{33.48}$$

и вычисляет один за другим члены соответствующей последовательности

$$x_1, x_2, x_3, x_4, \ldots . \tag{33.49}$$

(строка 7); переменная i указывает номер члена, начиная с 1ж первый член x_1 выбирается случайно (строки 1–2). (На самом деле в каждый момент хранится только один член последовательности, так что можно опустить индекс i и использовать одну переменную x для хранения текущего члена последовательности).

Переменная k последовательно принимает значения $2, 4, 8, \ldots$ (строки 4 и 13), когда значение i доходит до текущего значения k, значение x_i запоминается в переменной y, а переменная k увеличивается вдвое. Таким образом, y последовательно принимает значения $x_1, x_2, x_4, x_8, \ldots$

В строках 8–10 мы пытаемся найти делитель числа n, используя два члена последовательности — текущее значение x_i и сохранённое значение y. Эта попытка удачна, если число $d =$

Рисунок 33.7 33.7 ρ—эвристика Полларда. (**a**) Последовательность $x_{i+1} = (x_i^2 - 1) \bmod 1387$ с начальным элементом $x_1 = 2$. (Разложение: $1387 = 19 \cdot 73$.) Жирные стрелки соответствуют шагам цикла, выполняемым до нахождения делителя 19. (После этого выполнение процедуры прекращается — но на рисунке показан весь ρ-цикл.) Серым цветом показаны последовательные значения переменной y. Делитель 19 обнаруживается при сравнении $y = 63$ с $x_7 = 177$: $(63 - 177, 1387) = 19$. (**b**) Та же самая последовательность по модулю 19. По этому модулю упомянутые выше числа $x_7 = 177$ и $y = 63$ сравнимы. (**c**) Та же последовательность по модулю 73

gcd$(y - x_i, n)$ *отлично от* 1 *и от* n *(проверка выполняется в строке 9).*

Эти действия на первый взгляд выглядят странно, но по крайней мере легко видеть, что процедура Pollard-Rho *никогда не выдаёт неправильных ответов: уж если она чего напечатала, значит, это нетривиальный делитель* n*. Вопрос только в том, напечатает ли она хоть что-то. Сейчас мы приведём некоторые неформальные аргументы в пользу того, что если число* n *имеет нетривиальный делитель* s*, то есть шансы найти его после* $\sqrt{s}$ *проходов. (Напомним, что у составного числа* n *есть делитель, не превосходящий* $\sqrt{n}$*, так что ожидаемое число итераций можно оценить как* $\sqrt[4]{n}$*.*

Вот обещанные неформальные аргументы. Для начала заметим, что последовательность (33.49), начиная с какого-то места, становится периодической. (В самом деле, возможностей конечное число, так что должно встретиться повторение, а дальше уже всё пойдёт по циклу, так как следующий член последовательности зависит только от предыдущего.)

Поэтому последовательности можно изобразить в виде окружности с хвостиком (рис. 33.7): окружность соответствует периодической части последовательности, а хвостик — допериодической. (Эта картинка напоминает греческую букву ρ*, поэтому метод и называется* ρ*-эвристикой.)*

Что можно сказать о длине периода для случайно выбранного x_1*? Некоторая (пусть и безосновательная) оценка может быть получена так. Представим себе, что каждое* x_i *выбирается случайно и не зависит от предыдущих; сколько шагов пройдёт до первого повторения? (Конечно, предположение о случайности* x_i *абсурдно — каждый элемент есть функция предыдущего — но преобразование* $x \mapsto x^2 - 1$ *достаточно сложно, и с большой натяжкой его можно уподобить случайному перемешиванию.)*

Итак, сколько шагов пройдёт до первого повторения? Эту задачу мы уже рассматривали под названием парадокса с днями рождения (раздел 6.6.1), где отмечали, что среднее количество членов последовательности до первого повтора имеет порядок $\Theta(\sqrt{n})$*.*

Нас будут, впрочем, интересовать оценки длины периода не по модулю n, а по модулю делителей n. Пусть s — некоторый делитель n, для которого $gcd(s, n/s) = 1$ (например, выделим степени некоторого одного простого множителя в разложении n на множители). Посмотрим на остатки от деления чисел x_i на число s. Эта последовательность $x'_i = x_i \bmod s$ также можно рассматривать как заданную рекуррентным соотношением

$$x'_{i+1} = (x'^2_i - 1) \pmod{s}, \tag{33.50}$$

поскольку переход от модуля n к модулю s определён корректно (при $s \mid n$) и перестановочен с возведением в квадрат и вычитанием единицы.

Повторив приведённое выше вероятностное рассуждение, мы придём к заключению, что последовательность $\langle x'_i \rangle$ становится периодичной за $\Theta(\sqrt{s})$ шагов. Можно ожидать, что в последовательности $\langle x'_i \rangle$ повторение появится раньше, так как сравнимость по модулю n — более сильное условие, чем сравнимость по модулю s. (На рис. 33.7 как раз такой случай и показан.)

Теперь объясним, как используется переменная y. Пусть мы хотим найти повторяющиеся члены в последовательности $\langle x_i \rangle$. Для этого не нужно сравнивать каждый член с каждым — вместо этого можно запоминать члены $x_1, x_2, x_4, x_8, \ldots$ и сравнивать другие члены последовательности с ними, ища повторений. При этом, конечно, мы не гарантируем, что обнаружим самую раннюю пару повторяющихся членов. Но поскольку повторение, раз случившись, далее идёт по циклу, мы его не пропустим. При этом нам придётся просмотреть больше членов последовательности, чем если бы мы сравнивали все пары, но ненамного (не более чем в константу раз).

Процедура Pollard-Rho *сравнивает текущее значение x_i с запомненным значением y — но вместо того, чтобы проверять их равенство (и тем самым искать повторения в последовательности $\langle x_i \rangle$) вычисляет наибольший общий делитель их разности и числа n. Смысл этого в том, что тем самым можно уловить цикл в последовательности x'_i — если x_i и y сравнимы по модулю s, то наибольший общий делитель $x_i - y$ и n будет кратен s, и если только он не окажется равным n, то дело сделано - мы нашли нетривиальный делитель числа n. (Заметим ещё, что число s используется только в наших рассуждениях, не встречаясь в программе.)*

Всё может быть не так гладко, как мы описали, по двум причинам. Во-первых, наши оценки для числа шагов — всего лишь прикидки, и алгоритм может проработать намного дольше, прежде чем будет найден общий делитель. Во-вторых, обнаруженный делитель, кратный s, может оказаться равным n и, следовательно, *бесполезным. Такое возможно: пусть. например, $n = pq$, где p и q просты, и пусть последовательности $\langle x_i \bmod p\rangle$ и $\langle x_i \bmod q\rangle$ имеют одинаковые по длине периодические и допериодические части. Тогда повторения по модулю p совмещены с повторениями по модулю q, и в результате будет обнаруживаться общий делитель n.*

Обе эти возможности редко встречаются на практике. В крайнем случае можно изменить формулу, порождающую рекуррентную последовательность, положив, например, x_{i+1} равным $({x_i}^2 - c) \bmod n$. В качестве значения c не следует брать 0 и 2 (не будем сейчас объяснять, почему); другие значения вполне приемлемы.

Конечно, наш анализ весьма приблизителен, так как поскольку числа x_i не являются случайными — но на практике процедура работает хорошо, являясь одним из наиболее эффективных методов поиска небольших делителей больших целых чисел: оценка $\sqrt{s}$ на число операций зависит от величины делителя, а не от величины самого числа. При работе с β-битовым числом n мы ищем множители до $\sqrt{n}$, то есть размера не более $\beta/2$, и ожидаемое число арифметических операций имеет порядок $n^{1/4} = 2^{\beta/4}$ (что даёт порядка $n^{1/4}\beta^3 = 2^{\beta/4}\beta^3$ битовых операций.

Упражнения

33.9-1

Пользуясь рис. 33.7(a), определите, в какой моментэ процедура Pollard-Rho *напечатает делитель* 73 *числа* 1387.

33.9-2

Пусть задана функция $f : \mathbb{Z}_n \to \mathbb{Z}_n$ и значение $x_0 \in \mathbb{Z}_n$. Определим последовательность $x_1, x_2, \ldots$ соотношением $x_i = f(x_{i-1})$. Обозначим через u длину периодической части последовательности, а через t — длину допериодической части. Предложите эффективный алгоритм нахождения t и u. Оцените время его работы.

33.9-3

Через сколько шагов можно ожидать, что процедура Pollard-Rho *обнаружит делитель вида p^e, где p — простое число, а $e >$ 1?*

*33.9-4**

Процедура Pollard-Rho *на каждом шаге вычисляет наибольший общий делитель. Можно пытаться ускорить работу процедуры так: перемножить несколько соседних членов последовательности $\langle x_i\rangle$, и вычислить наибольший общих делитель их произведения и числа y. Разработайте соответствующую процедуру и объясните, почему она будет работать. Сколько соседних членов стоит брать при работе с β-битовыми числами?*

Задачи.

33-1 Двоичный алгоритм вычисления наибольшего общего дели-

теля.

Современные компьютеры выполняют операции сложения и вычитания, а также проверяют чётность числа быстрее, чем выполняют деление с остатком. Поэтому интересен двоичный алгоритм вычисления наибольшего общего делителя *(binary gcd algorithm), избегающий деления с остатком.*

a. Докажите, что $gcd(a, b) = 2gcd(a/2, b/2)$ *для чётных целых чисел* a *и* b.

b. Докажите, что если a *чётно, а* b *нечётно, то* $gcd(a, b) = gcd(a, b/2)$.

c. Докажите, что если оба числа a *и* b *нечётны, то* $gcd(a, b) = gcd((a - b)/2, b)$.

d. Постройте алгоритм, вычисляющий $gcd(a, b)$ *с использованием указанных соотношений за время* $O(\lg(\max(a, b)))$ *(считая, что вычитание, проверка на чётность, удвоение и и деление пополам выполняются за единицу времени).*

33-2 Оценка числа битовых операций в алгоритме Евклида.

a. Покажите, что обычный способ деления уголком числа a *на число* b *требует* $O((1+\lg q)\lg b)$ *битовых операций операций (*q *— частное).*

b. Положим $\mu(a, b) = (1 + \lg a)(1 + \lg b)$. *Докажите, что число битовых операций в процедуре* Euclid, *требуемых для сведения задачи вычисления* $gcd(a, b)$ *к задаче вычисления* $gcd(b, a \bmod b)$, *не превосходит* $c(\mu(a, b) - \mu(b, a \bmod b)$, *если* $c > 0$ *— достаточно большая константа. (Используйте оценку предыдущего пункта.)*

c. Докажите, что процедура Euclid(a, b) *выполняет не более* $O(\mu(a, b))$ *битовых операций. Докажите, что процедура* Euclid, *применённая к двум* β*−битовым числам, выполняет не более* $O(\beta^2)$ *битовых операций.*

33-3 Три алгоритма вычисления чисел Фибоначчи.

В этой задачи мы сравниваем три способа вычисления n*-го числа Фибоначчи* F_n *при заданном* n. *(Для простоты мы считаем, что операции сложения, вычитания или умножения выполняются за единичное время независимо от размера чисел.)*

a. Докажите, что время работы простейшей рекурсивной процедуры, непосредственно реализующей формулу (2.14), экспоненциально зависит от n.

b. Покажите, как техника запоминания промежуточных результатов позволяет сократить время до $O(n)$.

c. Покажите, как вычислить F_n *за время* $O(\lg n)$, *используя только операции (сложение и умножение) с целыми числами. (Указание. Рассмотрите матрицу*

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

и её степени.)

d. Оцените время работы трёх описанных алгоритмов, исходя из более реалистичных оценок времени сложения и умножения: считайте, что сложение двух β-битовых чисел требует времени $O(\beta)$, а умножение — $O(\beta^2)$.

33-4 Квадратичные вычеты.

Пусть p — нечётное простое число. Элемент $a \in \mathbb{Z}_p^*$ называется квадратичным вычетом *(quadratic residue), если существует $x \in \mathbb{Z}_p^*$, для которого $x^2 \equiv a \pmod p$.*

a. Докажите, что в группе $\mathbb{Z}_p^*$ имеется в точности $(p-1)/2$ квадратичных вычетов.

b. Пусть p — простое и $a \in \mathbb{Z}_p^*$. Символ Лежандра *(Legendre symbol), обозначаемый $\left(\frac{a}{p}\right)$, определяется так: $\left(\frac{a}{p}\right) = 1$, если a является квадратичным вычетом в $\mathbb{Z}_p^*$, и $\left(\frac{a}{p}\right) = -1$ в противном случае. Докажите, что*

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod p.$$

Предложите эффективный алгоритм, определяющий, является ли заданный элемент $a \in \mathbb{Z}_p^*$ квадратичным вычетом. Оцените время его работы.

c. Пусть простое число p имеет вид $4k+3$, а число $a \in \mathbb{Z}_p^*$ является квадратичным вычетом. Докажите, что $a^{k+1} \bmod p$ является квадратным корнем из a по модулю p. За какое время можно вычислить квадратный корень по модулю p из квадратичного вычета $a \in \mathbb{Z}_p^*$, если p имеет вид $4k+3$?

d. Предложите эффективный вероятностный алгоритм поиска (для любого простого p) элементов $a \in \mathbb{Z}_p^*$, не являющихся квадратичными вычетами. Сколько арифметических операций будет выполнять в среднем этот алгоритм?

Замечания.

Книга Нивена и Цукермана [191] — превосходный учебник по элементарной теории чисел. В книге Кнута [122] детально рассматриваются алгоритмы нахождения наибольшего общего делителя и другие базовые алгоритмы теории чисел. Из книг Ризеля [168] и Баха [16] можно узнать о современном состоянии вычислительной теории чисел. Задачи разложения на множители и проверки на простоту рассматриваются в книге Диксона [56]. В сборнике под редакцией Померанца [159] можно найти несколько хороших обзоров статей.

Кнут в [122] обсуждает происхождение алгоритма Евклида. Он содержится в седьмой книге знаменитых "Начал" Евклида (предложения 1 и 2), написанной около 300 г. до Р.Х. Возможно, идея алгоритма восходит к трудам Евдокса (около 375 г. до Р.Х.). Алгоритм Евклида претендует на роль древнейшего нетривиального *алгоритма — соперничать с ним может только так наываемый " русский народный алгоритм умножения чсиел" (глава 29), который был известен ещё древним египтянам.*

Кнут отмечает, что частный случай "китайской теоремы об остатках" был известен китайскому математику Сун-Цу, жившему где-то между 200 г. до Р.Х. и 200 г. от Р.Х. Тот же самый частный случай был разобран древнегреческим математиком Никомахом около 100 г. от Р.Х. Более общую форму теореме придал Чин Чью-Шао в 1247 году. В полной общности теорема была сформулирована и доказана Леонардом Эйлером в 1734 году.

Вероятностный алгоритм проверки чисел на простоту, рассмотренный нами, предложен Миллером [147] и Рабином [166]; он является (с точностью до постоянного множителя) самым быстрым из известных, Доказательство Теоремы 33.39, приведённое нами, следует идее Баха [15]. Более сильный результат о тесте Миллера — Рабина доказан Монье [148,149]. Использование случайности тут существенно — наилучший известный детерминированный (не использующий датчика случайных чисел) алгоритм проверки на простоту известен как версия Коэна — Ленстры [45] теста Адлемана, Померанца и Румели [3]. На проверку простоты числа n он тратит время $(\lg n)^{O(\lg\lg\lg n)}$ *(чуть больше полиномиального).*

Задача поиска большого "случайного" простого числа подробно обсуждют Бошемин, Брассар, Крепо, Готье и Померанц [20].

Идея криптосистемы с открытым ключом принадлежит Диффи и Хеллману [54]. Криптосистему RSA предложили в 1977 году Ривест, Шамир и Адлеман. С тех пор криптография стала быстро развивающейся областью со множеством интересных результатов. Например, Гольдвассер и Микэли [86], показали, как можно использовать рандомизацию (датчик случайных чисел) для построения криптосистем с открытым ключом вероятностные алгоритмы. Гольдвассер, Микэли и Ривест [87] предложили схему электронной подписи, взлом которой по трудности равносилен разложению чисел на множители. Гольдвассер, Микэли и Раков [87] ввели класс так называемых систем с нулевым разглашением, про которые можно доказать (при некоторых естественных допущениях), что ни одна из сторон не узнаёт больше, чем ей положено.

Метод разложения чисел, называемый ρ*-эвристикой, впервые был предложен Поллардом [156]. Вариант, излагаемый нами, предложил Брент [35].*

Время работы наилучшего известного алгоритма разложения числа n на множители возрастает с ростом n как экспонента, показателем которой является корень из длины n. В общем случае, возможно, наиболее эффективным является алгоритм " квадратичного решета", предложенный Померанцем в [158]. При не-

которых естественных предположениях время работы этого алгоритма можно оценить как $L(n)^{1+o(1)}$*, где* $L(n) = e^{\sqrt{\ln n \ln\ln n}}$*. Метод эллиптических кривых, предложенный Ленстрой [173], иногда эффективнее алгоритма квадратичного решета, поскольку он (так же как и* ρ*-алгоритм Полларда) быстро ищет небольшие делители; время поиска делителя* p *можно оценить как* $L(p)^{\sqrt{2}+o(1)}$.

34 Поиск подстрок

Большинство текстовых редакторов умеет искать заданное слово в редактируемом тексте — хочется, чтобы это происходило быстро. Другой пример задачи, в которой требуется искать заданную последовательность символов в строке, — поиск данной цепочки нуклеотидов в молекуле ДНК.

Говоря формально, задача поиска подстрок *(string-matching problem) состоит в следующем. Пусть даны "текст" — массив* $T[1..n]$ *длины* n *и "образец" — массив* $P[1..m]$ *длины* m*. Мы считаем, что элементы массивов* P *и* T *— символы некоторого конечного алфавита* Σ *(например,* $\Sigma = \{0,1\}$ *или* $\Sigma = \{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$*). Массивы, состоящие из символов алфавита* Σ*, часто называют* строками *(strings) символов, или* словами *в этом алфавите.*

Будем говорить, что образец P входит со сдвигом s *(occurs with shift s), или, эквивалентно,* входит с позиции $s+1$ *(occurs beginning at position* $s+1$*) в текст* T*, если* $0 \leqslant s \leqslant n-m$ *и* $T[s+1..s+m] = P[1..m]$ *(иными словами, если* $T[s+j] = P[j]$ *при* $1 \leqslant j \leqslant m$*). Если* P *входит со сдвигом* s *в текст* T*, то говорят, что* s — допустимый сдвиг *(valid shift), в противном случае* s — недопустимый сдвиг *(invalid shift). Задача поиска подстрок состоит в нахождении всех допустимых сдвигов для данных текста* T *и образца* P *(см. рис. 34.1).*

Настоящая глава посвящена различным алгоритмам для поиска подстрок. В разделе 34.1 мы рассматриваем простейший алгоритм, работающий за время $O((n-m+1)m)$ *(в худшем случае). Затем в разделе 34.2 мы рассказываем об алгоритме Рабина — Карпа. Этот остроумный алгоритм поиска подстрок также в*

Рис.34.1. Переводы слов на рисунке: text — текст, pattern — образец.

Подпись:

Задача поиска подстрок. Требуется найти все вхождения образца $P = \mathsf{abaa}$ в текст $T = \mathsf{abcabaabcabac}$. Образец входит в текст только один раз, со сдвигом $s = 3$ (стало быть, 3 — допустимый сдвиг).

Рис. 34.2
Подпись: Рис. 34.2. Доказательство леммы 34.1. Совпадающие символы соединены вертикальными линиями, а совпадающие части строк заштрихованы. Рис. (а) соответствует случаю $|x| \leqslant |y|$, рис. (б) — случаю $|x| \geqslant |y|$, рис. (в) — случаю $|x| = |y|$.

худшем случае работает за время $O((n-m+1)m)$, но на практике он в среднем гораздо быстрее; кроме того, алгоритм Рабина — Карпа обобщается на другие задачи поиска образца. В разделе 34.3 мы описываем алгоритм поиска подстрок, который по заданному образцу строит конечный автомат, и затем пропускает через этот автомат текст T. Время работы алгоритмов, основанных на этой идее, может быть доведено до $O(n + m|\Sigma|)$. Аналогичный, но более изощрённый алгоритм Кнута — Морриса — Пратта (сокращенно КМР) работает за время $O(m + n)$; этому алгоритму посвящен раздел 34.4. Наконец, в разделе 34.5 описывается алгоритм Бойера-Мура. Этот алгоритм зачастую оказывается наиболее удобным на практике, хотя, подобно алгоритму Рабина-Карпа, в худшем случае он не дает выигрыша по сравнению с простейшим алгоритмом.

34.0.1 Обозначения и терминология

Через Σ^* обозначается множество всех конечных строк над алфавитом Σ, включая пустую строку *(empty string), имеющую нулевую длину и обозначаемую ε. Длина строки x обозначается $|x|$.* Соединение, *или* конкатенация *(concatenation) строк x и y получится, если выписать строку x, а за ней встык — строку y. Конкатенация строк x и y обозначается xy; очевидно, $|xy| = |x| + |y|$.*

Мы будем говорить, что строка w — префикс *(prefix), или* начало, *строки x, если $x = wy$ для некоторого $y \in \Sigma^*$. Будем говорить, что w —* суффикс *(suffix), или* конец, *строки x, если $x = yw$ для некоторого $y \in \Sigma^*$. Будем писать $w \sqsubset x$, если w — префикс x, и $w \sqsupset x$, если w — суффикс x. Например,* $\mathsf{ab} \sqsubset \mathsf{abcca}$ *и* $\mathsf{cca} \sqsupset \mathsf{abcca}$. *[С этими обозначеними нужно обращаться осторожно: $a \sqsubset b$ и $b \sqsupset a$ означают совершенно разное!]*

Пустая строка является префиксом и суффиксом любой строки; если w — префикс или суффикс x, то $|w| \leqslant |x|$. Для любых строк x и y и для любого символа a соотношения $x \sqsupset y$ и $xa \sqsupset ya$ равносильны; отношения $\sqsubset$ и $\sqsupset$ транзитивны. В дальнейшем мы будем пользоваться следующей леммой.

Лемма 34.1 (Лемма о двух суффиксах)

Пусть x, y и z — строки, для которых $x \sqsupset z$ и $y \sqsupset z$. Тогда $x \sqsupset y$, если $|x| \leqslant |y|$; $y \sqsupset x$, если $|x| \geqslant |y|$, и $x = y$, если $|x| = |y|$.

Доказательство. См. рис. 34.2.

Если $S[1..r]$ — строка длины r, то ее префикс длины $k \leqslant r$ будет обозначаться $S_k = S[1..k]$ (в частности, $S_0 = \varepsilon$ и $S_r = S$). В этих обозначениях задача о нахождении образца P длины m в тексте T длины n состоит в нахождении всех таких s из промежутка $0 \leqslant s \leqslant n - m$, что $P \sqsupset T_{s+m}$.

При записи алгоритмов для поиска подстрок мы будем рассматривать проверку равенства двух строк как элементарную операцию, время выполнения которой пропорционально длине сравниваемых строк. Если сравнивать строки слева направо и останавливаться, как только обнаружено расхождение, то стоимость сравнения строк x и y есть $\Theta(t+1)$, где t — длина наибольшего общего префикса строк x и y. (Мы пишем $t+1$ вместо t, учитывая сравнение первых не совпавших символов.)

34.1 Простейший алгоритм

Первый приходящий в голову алгоритм для поиска образца P в тексте T последовательно проверяет равенство $P[1..m] = T[s+1..s+m]$ для всех $n-m+1$ возможных значений s:

```
Naive-String-Matcher(T,P)
1 n \gets length[T]
2 m \gets length[P]
3 for s \gets 0 to n-m
4     do if P[1..m]= T[s+1..s+m]
5             then print ``Подстрока входит со сдвигом ''s
```

Можно сказать, что мы двигаем образец вдоль текста и проверяем все его положения (рис. 34.3). Отметим, что проверка в строке 4 представляет собой ещё один цикл.

Время работы процедуры Naive-String-Matcher *в худшем случае есть $\Theta((n-m+1)m)$. В самом деле, пусть $T = \mathtt{a}^n$ (буква* a*, повторённая n раз), а $P = \mathtt{a}^m$. Тогда для каждой из $n-m+1$ проверок (строка 4) будет выполнено m сравнений символов, всего $(n-m+1)m$, что есть $\Theta(n^2)$ (при $m = \lfloor n/2 \rfloor$).*

Рис. 34.3
Подпись:
Рис. 34.3. Простейший алгоритм ищет образец $P = \mathtt{aab}$ в тексте $T = \mathtt{acaabc}$. Четыре последовательные попытки изображены на рис. (а)–(г). Буквы, для которых сравнение прошло успешно, соединены и показаны серым. Буквы, на которых выявлено несовпадение, соединены ломаными линиями. При этом $s = 2$ — единственный допустимый сдвиг.

Простейший алгоритм — не лучший (далее мы расскажем об алгоритме, работающем за время $O(n+m)$). Неэффективность простейшего алгоритма связана с тем, что информация о тексте T, получаемая при проверке данного сдвига s, никак не используется при проверке последующих сдвигов. Между тем такая информация может очень помочь. Пусть, например, $P = \mathsf{aaab}$ и мы выяснили, что сдвиг $s = 0$ допустим. Тогда сдвиги 1, 2 и 3 заведомо недопустимы, поскольку $T[4] = \mathsf{b}$. Далее мы обсудим различные способы реализации этой идеи.

Упражнения

34.1-1

Какие сравнения символов делает простейший алгоритм при $P = \mathsf{0001}$ и $T = \mathsf{000010001010001}$?

34.1-2

Покажите, что в худшем случае простейший алгоритм найдёт первое вхождение подстроки за время $\Theta((n-m+1)(m-1))$.

34.1-3

Пусть все символы в образце P различны. Как усовершенствовать алгоритм Naive-String-Matcher, *чтобы он работал за время $O(n)$, где n — длина текста?*

34.1-4

Пусть алфавит содержит d символов, и пусть образец и текст — случайные строки длины m и n соответственно. Покажите, что математическое ожидание числа сравнений символов, производимых простейшим алгоритмом при выполнении строки 4, есть

$$(n-m+1)\frac{1-d^{-m}}{1-d^{-1}} \leqslant 2(n-m+1)$$

(сравнение строк прекращается, как только найдены несовпадающие символы или когда просмотрен весь образец). Таким образом, для случайных строк простейший алгоритм вполне эффективен.

34-1.5 Пусть в образце (но не в тексте!) может встречаться, наряду с символами из алфавита Σ, символ $\Diamond$, называемый символом пропуска *(gap character), который соответствует произвольной подстроке (в том числе пустой). Например, образец $\mathsf{ab}\Diamond\mathsf{ba}\Diamond\mathsf{c}$ входит в текст $\mathsf{cabccbacbacab}$ и как*

$$\mathsf{c}\underbrace{\mathsf{ab}}_{\mathsf{ab}}\underbrace{\mathsf{cc}}_{\Diamond}\underbrace{\mathsf{ba}}_{\mathsf{ba}}\underbrace{\mathsf{cba}}_{\Diamond}\underbrace{\mathsf{c}}_{\mathsf{c}}\mathsf{ab},$$

и как

$$\mathsf{c}\underbrace{\mathsf{ab}}_{\mathsf{ab}}\underbrace{\mathsf{ccbac}}_{\Diamond}\underbrace{\mathsf{ba}}_{\mathsf{ba}}\underbrace{\qquad}_{\Diamond}\underbrace{\mathsf{c}}_{\mathsf{c}}\mathsf{ab}.$$

Разработайте полиномиальный алгоритм, выясняющий, входит ли данный образец (с символами пропуска) в данный текст.

34.2 Алгоритм Рабина — Карпа

Рабин и Карп изобрели алгоритм поиска подстрок, который эффективен на практике и к тому же обобщается на другие аналогичные задачи (например, поиск образца на двумерной решётке). Хотя в худшем случае время работы алгоритма Рабина-Карпа есть $\Theta((n-m+1)m)$*, в среднем он работает достаточно быстро.*

Предположим для начала, что $\Sigma = \{\mathtt{0}, \mathtt{1}, \mathtt{2}, \ldots, \mathtt{9}\}$ *(в общем случае можно считать, что каждый символ в алфавите* Σ *есть* d*-ичная цифра, где* $d = |\Sigma|$*). Тогда строку из* k *символов можно рассматривать как десятичную запись числа (*k*-значного), а сами символы — как цифры.*

Если $P[1..m]$ *— образец, то через* p *обозначим число, десятичной записью которого он является; аналогично, если* $T[1..n]$ *— текст, то для* $s = 0, 1, \ldots, n-m$ *обозначим через* t_s *число, десятичной записью которого является строка* $T[s+1..s+m]$*. Очевидно,* s *является допустимым сдвигом тогда и только тогда, когда* $t_s = p$*. Если бы мы могли вычислить* p *за время* $O(m)$ *и все* t_i *за время* $O(n)$ *(временно закроем глаза на то обстоятельство, что эти вычисления могут привести к слишком большим числам), то мы смогли бы найти все допустимые сдвиги за время* $O(n)$*, сравнивая* p *со всеми* t_s *по очереди.*

Вычислить p *за время* $O(m)$ *действительно можно, по схеме Горнера (разд. 32.1):*

$$p = P[m] + 10(P[m-1] + 10(P[m-2] + \cdots + 10(P[2] + 10P[1])\ldots)).$$

Точно так же за время $O(m)$ *можно вычислить* t_0*.*

Чтобы вычислить $t_1, t_2, \ldots, t_{n-m}$ *за время* $O(n-m)$*, заметим, что при известном* t_s *можно вычислить* t_{s+1} *за время* $O(1)$*. В самом деле,*

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1] : \qquad (34.1)$$

чтобы получить строку $T[s+2..s+m+1]$ *из* $T[s+1..s+m]$*, надо удалить* $T[s+1]$ *(то есть вычесть* $10^{m-1}T[s+1]$ *из* t_s*) и приписать справа* $T[s+m+1]$ *(то есть умножить полученную разность на* 10 *и прибавить к ней* $T[s+m+1]$*). Если вычислить константу* 10^{m-1} *заранее (с помощью техники, описанной в разд. 33.6, это можно сделать за время* $O(\lg m)$*; впрочем, оценка не ухудшится, если непосредственно перемножить* $m-1$ *десятку за время* $O(m)$*), то* стоимость вычислений по формуле (34.1) ограничена сверху константой; стало быть, числа p и $t_0, t_1, \ldots, t_{n-m}$ могут быть найдены за время $O(n+m)$, и также за время $O(n+m)$ могут быть найдены все вхождения образца $P[1..m]$ в текст $T[1..n]$.

До сих пор мы не учитывали того, что числа p и t_s велики — настолько, что предположение о выполнении арифметических операций за время $O(1)$ едва ли допустимо. К счастью, эту трудность можно обойти следующим образом: надо проводить вычисления чисел p и t_0, а также вычисления по формуле (34.1), по модулю фиксированного числа q (рис. 34.4; по поводу арифметики по модулю q см. разд. 33.1). Тогда все числа не превосходят q и можно считать, что число p и все t_j будут действительно вычислены за время $O(n+m)$. Обычно в качестве q выбирают простое число, для которого $10q$ помещается в машинное слово — это упрощает программирование вычислений. В общем случае (для алфавита $\{\,\mathtt{0}, \mathtt{1}, \mathtt{2}, \ldots, \mathtt{d}\,\}$) выбирают такое простое число q, что dq помещается в машинное слово (благодаря этому все арифметические операции происходят в пределах одного слова); рекуррентное соотношение (34.1) приобретает вид

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q, \qquad (34.2)$$

где $h \equiv d^{m-1} \pmod q$.

Вычисления по модулю q хороши всем, кроме одного: из равенства $t_s \equiv p \pmod q$ ещё не следует, что $t_s = p$. Поэтому, если $t_s \not\equiv p \pmod q$, то сдвиг s заведомо недопустим и о нем можно забыть, а если $t_s \equiv p \pmod q$, то надо еще проверить, совпадают ли строки $P[1..m]$ и $T[s+1..s+m]$ на самом деле. Если совпадают, то мы нашли вхождение образца в строку, а если не совпадают, то произошло холостое срабатывание (spurious hit). Если простое число q достаточно велико, то можно надеяться, что дополнительные затраты на анализ холостых срабатываний будут невелики.

Запишем текст соответствующей процедуры Rabin-Karp-Matcher. Она получает на вход текст T, образец P, "основание системы счисления" d (обычно берут $d = |\Sigma|$) и простое число q.

```
Rabin-Karp-Matcher(T,P,d,q)
n \gets length[T]
m \gets length[P]
h \gets d^{m-1} \bmod q
p \gets 0
t \gets 0
for i \gets 1 to m
    do p \gets (dp+P[i]) \bmod q
       t \gets (dt+T[i]) \bmod q
for s \gets 0 to n-m
```

Рис.34.4, занимающий целую страницу.
Переводы надписей, входящих в рисунок: valid match — вхождение образца, spurious hit — холостое срабатывание, old high-order digit — цифра старшего разряда, new low-order digit — цифра младшего разряда, shift не переводить и не воспроизводить (ни слово, ни стрелку).

Подпись:
Рис.34.4. Алгоритм Рабина-Карпа. $\Sigma = \{\,\mathtt{0}, \mathtt{1}, \mathtt{2}, \ldots, \mathtt{9}\,\}$, $q = 13$. (а) Строка T (текст). Серым выделено окошко ширины 5. Численное значение выделенной подстроки равно 7 по модулю 13. (б) Для того же текста указаны численные значения (по модулю 13) всех подстрок длины 5. Если образец есть $P = \mathtt{31415}$, то нас интересуют подстроки со значением 7, поскольку $31415 \equiv 7 \pmod{13}$. Таких подстрок всего две; одна из них соответствует вхождению образца в текст, а другая — холостому срабатыванию. (в) Изменение числового значения при сдвиге окошка. В предыдущем окошке стояло 31415. Удалив цифру старшего разряда и приписав новую цифру младшего разряда, получаем 14152. Те же вычисления по модулю 13 из старого значения 7 получают новое значение 8.

```
10     do if p=t
11           then if P[1..m]=T[s+1..s+m]
12                 then print ``Образец входит со сдви-
гом ''s
13        if s<n-m
14           then t \gets (d(t-T[s+1]h) + T[s+m+1]) \bmod q
```

Опишем работу процедуры. Все символы рассматриваются как d-ичные цифры. В строках 1–5 переменным присваиваются начальные значения (h — это "единица старшего разряда" в d-ичной системе). В цикле в строках 6–8 с помощью схемы Горнера вычисляются значения p и t_0 (последнее присваивается переменной t). Цикл в строках 9–14 перебирает все возможные значения s; в момент исполнения строки 10 имеем $t \equiv t_s \pmod q$. Если оказывается, что $t_s \equiv p$, то строки $T[s+1..s+m]$ и $P[1..m]$ сравниваются и, в случае совпадения, об этом печатается сообщение (строки 11–12). Если $t_s \not\equiv p$, то программа проверяет, будет ли цикл выполняться далее (строка 13), и если будет, то обновляет значение t по формуле (34.2) — строка 14.

В худшем случае эта процедура требует времени $\Theta((n-m+1)m)$, как и простейший алгоритм — уже потому, что для всех допустимых сдвигов происходит посимвольная проверка. Например, если $P = \mathtt{a}^m$ и $T = \mathtt{a}^n$, то сравнение в строке 11 будет выполняться для всех значений s (и алгоритм отличается от тривиального лишь в худшую сторону за счёт дополнительных затрат на вычисление h в строке 3, на цикл в строках 6–8 и на вычисления в строке 14).

Во многих приложениях следует ожидать, что допустимых сдвигов будет немного; в этом случае время работы алгоритма Рабина — Карпа будет $O(n+m)$ плюс небольшие дополнительные затраты на обработку холостых срабатываний. Можно сделать нестрогую прикидку, основываясь на следующих соображениях. Будем считать, что отображение редукции по модулю q — случайная функция из Σ^* в $\mathbf{Z}_q$. Это утверждение трудно поддается формализации и доказательству, но эмпирически подтверждается (ср. разд. 12.3.1, где мы обсуждали применение деления с остатком к хешированию). Тогда можно надеяться, что количество холостых срабатываний есть $O(n/q)$, поскольку вероятность того, что случайное число t_s сравнимо с p по модулю q, равна $1/q$. Стало быть, ожидаемое время работы алгоритма Рабина — Карпа есть

$$O(n) + O(m(v + n/q)),$$

где v — количество вхождений образца в текст. Если $q \geqslant m$ (то есть длина образца не превосходит выбранного значения q) и $v = O(1)$, то получается, что алгоритм работает за время $O(n+m)$.

34.2.1 Упражнения

34.2-1

Сколько холостых срабатываний даст алгоритм Рабина-Карпа, если $q = 11$, $T = 3141592653589793$ и $P = 26$?

34.2-2 Обобщите алгоритм Рабина — Карпа на случай, когда в тексте надо искать одну из k данных подстрок.

34-2.3 Обобщите алгоритм Рабина — Карпа на случай, когда надо искать квадрат размером $m \times m$ в матрице размером $n \times n$ с заданным содержимым (т.е. получающийся из образца параллельным переносом).

34-2.4 У Ани на компьютере есть n-битовый файл $A = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$, а у Бори — n -битовый файл $B = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$. Они хотят проверить, идентичны ли эти файлы, не пересылая их друг другу, следующим образом. Выбирается простое число $q > 1000n$ и случайное целое число $x \in \{0, 1, \ldots, q-1\}$. После этого Аня вычисляет выражение

$$A(x) = \left(\sum_{i=0}^{n} a_i x^i \right) \bmod q,$$

а Боря вычисляет аналогичное выражение $B(x)$. Покажите, что, если $A \neq B$, существует лишь один шанс из тысячи, что $A(x) = B(x)$ (и уж конечно $A(x) = B(x)$, если $A = B$). (Указание: воспользуйтесь упражнением 33.4-4).

34.3 Поиск подстрок с помощью конечных автоматов

Многие алгоритмы для поиска подстрок начинают с того, что строят конечный автомат, который находит в тексте T все вхождения образца P. В этом разделе мы опишем, как можно построить такой автомат. Сам по себе поиск подстроки с помощью конечного автомата весьма эффективен: каждый символ поступает на вход конечного автомата только единожды, а обработка каждого символа занимает ограниченное время, так что общее время работы есть $\Theta(n)$ — после того, однако, как автомат построен! К сожалению, время на построение конечного автомата может быть велико, если велик алфавит Σ (в разд. 34.4 мы обсудим остроумный способ обойти эту трудность).

В этом разделе мы дадим определение конечного автомата, затем рассмотрим специальный конечный автомат, предназначенный для поиска подстрок, и наконец покажем, как сконструировать этот автомат, исходя из подстроки, которую он призван искать.

Рис. 34.5. Переводы слов в рисунке: input — вход, state — состояние.

Подпись:

Конечный автомат со множеством состояний $Q = \{0, 1\}$, начальным состоянием $q_0 = 0$ и входным алфавитом $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. (а) Таблица значений функции перехода δ. (б) Функция перехода в виде диаграммы. Состояние 1 — единственное допускающее состояние (чёрное). Стрелками показаны переходы. Например стрелка из состояния 1 в состояние 0, помеченная буквой b, означает, что $\delta(1, \mathtt{b}) = 0$. Этот автомат допускает строки, оканчивающиеся на нечётное число букв a (точнее говоря, строки вида $y\mathtt{a}^k$, где строка y пуста или оканчивается на b, а число k нечётно). Например, для входной строки abaaa последовательность состояний (включая исходное) будет $\langle 0, 1, 0, 1, 0, 1 \rangle$, и эта строка допускается; для входной строки abbaa последовательность состояний будет $\langle 0, 1, 0, 0, 1, 0 \rangle$, и эта строка отвергается.

34.3.1 Конечные автоматы

По определению, конечный автомат *(finite automaton) — это пятерка $M = (Q, q_0, A, \Sigma, \delta)$, где:*

- Q *— конечное множество* состояний *(states);*
- $q_0 \in Q$ — начальное состояние *(start state);*
- $A \subseteq Q$ *— конечное множество* допускающих состояний *(accepting states);*
- Σ *— конечный* входной алфавит *(input alphabet);*
- δ *— функция из $Q \times \Sigma$ в Q, называемая* функцией перехода *(transition function) автомата.*

Первоначально конечный автомат находится в состоянии q_0; затем он по очереди читает символы из входной строки. Находясь в состоянии q и читая символ a, автомат переходит в состояние $\delta(q, a)$. Если автомат находится в состоянии $q \in A$, говорят, что он допускает *(accepts) прочитанную часть входной строки; если же $q \notin A$, то прочитанная часть строки* отвергнута *(is rejected). На рис. 34.5 показан пример простого автомата с двумя состояниями.*

С конечным автоматом M связана функция $\varphi\colon \Sigma^* \to Q$, называемая функцией конечного состояния *(final-state function), определяемая следующим образом: $\varphi(w)$ есть состояние, в которое придёт автомат (из начального состояния), прочитав строку w. Автомат допускает строку w тогда и только тогда, когда $\varphi(w) \in A$. Функцию φ можно определить рекуррентно:*

$$\begin{aligned} &\varphi(\varepsilon) = q_0; \\ &\varphi(wa) = \delta(\varphi(w), a) \qquad \text{для любых } w \in \Sigma^* \text{ и } a \in \Sigma. \end{aligned}$$

34.3.2 Автоматы для поиска подстрок

Для каждого образца P можно построить конечный автомат, ищущий этот образец в тексте (см. рис. 34.6, где изображен автомат, соответствующий образцу $P =$ ababaca*). Зафиксируем до конца этого раздела строку-образец P.*

Первым шагом в построении автомата, соответствующего строке-образцу $P[1..m]$, будет построение по P вспомогательной функции $\sigma\colon \Sigma^* \to \{\,0, 1, \ldots, m\,\}$, называемой суффикс-функцией *(suffix function). По определению, σ сопоставляет строке x длину максимального суффикса x, являющегося префиксом P:*

$$\sigma(x) = \max\{\,k : P_k \sqsupset x\,\}.$$

Поскольку $P_0 = \varepsilon$ является суффиксом любой строки, σ определена на всем Σ^*. Пример: если $P =$ ab*, то $\sigma(\varepsilon) = 0$, $\sigma(\text{ccaca}) = 1$, $\sigma(\text{ccab}) = 2$. Если длина P равна m, то $\sigma(x) = m$ тогда и только тогда, когда P — суффикс x. Если $x \sqsupset y$, то $\sigma(x) \leqslant \sigma(y)$.*

Теперь определим конечный автомат, соответствующий образцу $P[1..m]$, следующим образом:

- *Множество состояний есть $Q = \{\,0, 1, \ldots, m\,\}$. Начальное состояние $q_0 = 0$, единственное допускающее состояние есть m.*
- *Функция перехода δ определена следующей формулой (q — состояние, $a \in \Sigma$ — символ):*

$$\delta(q, a) = \sigma(P_q a). \tag{34.3}$$

Объясним, откуда берётся формула (34.3). Мы хотим сконструировать автомат таким образом, чтобы при его действии на строку T соотношение

$$\varphi(T_i) = \sigma(T_i) \tag{34.4}$$

являлось инвариантом (тогда равенство $\varphi(T_i) = m$ будет равносильно тому, что P входит в T со сдвигом $i - m$, и автомат тем самым найдёт все допустимые сдвиги). Но в этом случае вычисление перехода по формуле (34.3) необходимо для поддержания истинности инварианта (см. теорему 34.4 ниже).

Например, в автомате рис. 34.6, имеем $\delta(5, \text{b}) = 4$: если $q = 5$, прочитанная часть входа кончается на ababa*, и после добавления входного символа* b *наибольший суффикс прочитанной части, являющийся префиксом P, будет равен* abab.

Запишем действие конечного автомата, ищущего подстроку P длины m в данном тексте T, в виде программы (δ обозначает функцию перехода):

```
Finite-Automaton-Matcher(T,\delta,m)
```

Рис. 34.6, занимающий целую страницу. Переводы слов, входящих в рисунок: input — вход, state — состояние.

Подпись:

Рис. 34.6. (а) Таблица переходов для конечного автомата, допускающего строки, оканчивающиеся на ababaca (и только их). Здесь 0 — исходное состояние, 7 — единственное допускающее состояние (зачернено). Если из i в j ведет стрелка, помеченная буквой a, это означает, что $\delta(a, i) = j$. Жирные стрелки, идущие слева направо, соответствуют успешным этапам поиска подстроки P [если мы в состоянии j, то j последних прочитанных букв текста совпадают с j первыми буквами образца; если мы перешли из состояния j в состояние $j+1$, то очередная буква текста также совпадает с очередной буквой образца — шансы найти образец растут!]. Стрелки, идущие справа налево, соответствуют неудачам [последние j букв текста совпадали с первыми j буквами образца, но очередная буква — не такая, как хотелось бы]. Не все стрелки, идущие справа налево, показаны на рисунке: если из состояния i не выходит стрелки, помеченной буквой a, то подразумевается, что $\delta(i, a) = 0$. (б) Таблица переходов для того же автомата. Клеточки, соответствующие успешным этапам поиска (жирным стрелкам на диаграмме), выделены серым. (в) Результат применения автомата к тексту $T =$ abababacaba. Под каждым символом $T[i]$ записано состояние автомата после прочтения этого символа (иными словами, значение $\varphi(T_i)$). Найдено одно вхождение образца (начиная с позиции 3).

Рис. 34.7
Подпись:
К доказательству леммы 34.2: если $r = \sigma(xa)$, то $r \leqslant \sigma(x) + 1$.

Рис. 34.8
Подпись:
К доказательству леммы 34.3. Из рисунка видно, что $r = \sigma(P_q a)$, где $q = \sigma(x)$ и $r = \sigma(xa)$.

```
n \gets length[T]
q \gets 0
for i \gets 1 to n
    do q \gets \delta(q, T[i])
       if q=m
          then s \gets i-m
               print ‘‘Образец входит со сдвигом ’’s
```

Поскольку эта программа обрабатывает каждый символ из текста T по разу, время её работы есть $O(n)$. Однако следует учесть и время, требуемое для вычисления функции перехода δ. Мы этим вскоре займёмся, но сначала докажем, что процедура FINITE-AUTOMATON-MATCHER *правильно находит все вхождения подстроки T.*

Как отмечалось выше, нам достаточно показать, что для всех i выполнено соотношение (34.4), то есть что после прочтения символа $T[i]$ автомат оказывается в состоянии $\sigma(T_i)$. Это вытекает из следующих двух лемм.

Лемма 34.2 (неравенство для суффикс-функции)

Для любых строки x и символа a имеем $\sigma(xa) \leqslant \sigma(x) + 1$.

Доказательство.

Если $\sigma(xa) > \sigma(x) + 1$, то отбросим последний символ a от наибольшего суффикса xa, являющегося префиксом P, и получим суффикс строки x, имеющий длину больше $\sigma(x)$ и являющийся префиксом P — противоречие (см. рис. 34.7).

Лемма 34.3 (Рекуррентная формула для суффикс-функции)

Пусть $q = \sigma(x)$, где x — строка. Тогда для любого символа a имеем $\sigma(xa) = \sigma(P_q a)$.

Доказательство.

Лемма 34.2 гласит, что $\sigma(xa) \leqslant q + 1$. Поэтому значение $\sigma(xa)$ не изменится, если оставить от строки xa последние $q + 1$ символов, то есть заменить его на строку $P_q a$ (напомним, что последние q символов строки x образуют слово P_q, так как $\sigma(x) = q$ (рис. 34.8).

Из леммы 34.3 немедленно вытекает

Теорема 34.4

Пусть φ — функция конечного состояния автомата для поиска *подстроки $P[1..m]$. Если $T[1..n]$ — произвольный текст, то*

$$\varphi(T_i) = \sigma(T_i)$$

для $i = 0, 1, \ldots, n$.

Доказательство.

Для $i = 0$ это соотношение очевидно. Лемма 34.3 и формула (34.3) для функции перехода показывают, что оно сохраняется при прочтении автоматом очередного символа.

В силу доказанной теоремы, автомат после прочтения i символов текста находится в состоянии q тогда и только тогда, когда P_q является самым длинным суффиксом строки T_i, являющимся одновременно префиксом строки P. В частности, $q = m$ означает, что автомат только что прочёл подстроку P. Это доказывает правильность алгоритма Finite-Automaton-Matcher.

34.3.3 Вычисление функции перехода

Функцию перехода δ, соответствующую образцу $T[1..m]$, можно вычислить так:

```
Compute-Transition-Function(P,\Sigma)
1 m \gets length[P]
2 for q \gets 0 to m
3     do for (для) всех символов a \in \Sigma
4        do k \gets \min(m+1, q+2)
5           repeat k \gets k-1
6            until P_k \sqsupset P_qa
7           \delta(q,a) \gets k
8 return \delta
```

Эта процедура вычисляет функцию δ "в лоб": циклы, начинающиеся в строках 2 и 3, перебирают все пары (q, a), а в строках 4–7 наибольшее значение k, при котором для данной пары (q, a) выполнено соотношение $P_k \sqsupset P_q a$, находится прямым перебором, начиная с наибольшего априори возможного значения k, то есть $\min(m, q+1)$.

Время работы этого алгоритма есть $O(m^3|\Sigma|)$: в самом деле, два внешних цикла дают множитель $m|\Sigma|$, внутренний цикл **repeat** *может выполняться не более $m+1$ раз, и сравнение в строке 6 требует $O(m)$ операций. На самом деле функцию перехода можно вычислить гораздо быстрее, за время $O(m|\Sigma|)$ (см. упр. 34.4-6). В этом случае время поиска образца длины m в тексте длины n будет $O(n + m|\Sigma|)$.*

Упражнения

34.3-1

Постройте автомат для поиска подстроки $P =$ aabab и продемонстрируйте его работу на тексте $T =$ aaababaabaababaab.

34.3-2

Нарисуйте диаграмму переходов автомата для поиска подстроки $P =$ ababbabbababbababbabb (над алфавитом $\Sigma = \{\mathtt{a}, \mathtt{b}\}$).

34.3-3

Будем говорить, что P — строка с уникальными префиксами *(P is nonoverlappable), если соотношение $P_k \sqsupset P_q$ возможно лишь при $k = 0$ или $k = q$. Как выглядит диаграмма переходов автомата для поиска подстроки с уникальными префиксами?*

*34.3-4**

Даны два образца P и P'. Постройте конечный автомат, находящий все вхождения каждого из этих образцов в данный текст. Постарайтесь, чтоб число состояний вашего автомата было поменьше.

34.3-5

Пусть образец P содержит, наряду с символами из алфавита Σ, еще и символы пропусков (упражнение 34.1-5). Постройте конечный автомат, который отыскивает все вхождения такого образца P в текст T за время $O(|T|)$.

34.4 Алгоритм Кнута — Морриса — Пратта

Теперь мы переходим к алгоритму для поиска подстрок, работающему за линейное время. Этот алгоритм, предложенный Кнутом, Моррисом и Праттом, работает за время $\Theta(m+n)$. Такое ускорение достигается за счет того, что предварительно вычисляется не функция перехода $\delta[0..m, 1..|\Sigma|]$, а в $|\Sigma|$ раз меньший массив — "префикс-функция" $\pi[1..m]$ (её вычисление производится за время $O(m)$). Зная функцию π, можно вычислить $\delta(q,a)$ для любого состояния $q \in \{0, 1, \ldots, m\}$ и символа a с учётной стоимостью $O(1)$ (в смысле амортизационного анализа). Сейчас мы увидим, как это делается.

34.4.1 Префикс-функция, ассоциированная с образцом

Префикс-функция, ассоциированная с образцом P, несёт информацию о том, где в строке P повторно встречаются различные префиксы этой строки. Использование этой информации позволяет избежать проверки заведомо недопустимых сдвигов (говоря в терминах простейшего алгоритма поиска) или обойтись без *предварительного вычисления функции перехода (в терминах конечных автоматов).*

К префикс-функции приводит следующий ход мыслей. Пусть простейший алгоритм ищет вхождения подстроки $P = \mathtt{ababaca}$ в текст T. Предположим, что для некоторого сдвига s оказалось, что q первых символов образца совпадают с символами текста, а в следующем символе имеется расхождение (рис. 34.9 (а), где $q = 5$). Стало быть, мы знаем q символов текста, от $T[s+1]$ до $T[s+q]$, и из этой информации можно заключить, что некоторые последующие сдвиги будут заведомо недопустимы. В примере на рис. 34.9, скажем, сразу видно, что недопустим сдвиг $s+1$, поскольку при этом сдвиге первый символ образца (буква a*) окажется напротив $s+2$-го символа текста, совпадающего со вторым символом образца, — а это буква* b*. А вот при сдвиге на $s+2$ (рис. 34.9 (б)) первые три символа образца совпадают с тремя последними из известных нам символов текста, так что этот сдвиг априори отбросить нельзя. В общем случае хотелось бы уметь отвечать на такой вопрос:*

> *Пусть $P[1..q] = T[s{+}1..s{+}q]$; каково наименьшее значение сдвига $s' > s$, для которого*
>
> $$P[1..k] = T[s'+1..s'+k], \tag{34.5}$$
>
> *где $s'+k = s+q$?*

Число s' — это наименьшее значение сдвига, большее s, которое нельзя отбросить с порога, исходя из равенства $T[s+1..s+q] = P[1..q]$. Больше всего нам повезёт, если $s' = s{+}q$: тогда мы можем не рассматривать сдвиги $s{+}1, s{+}2, \ldots, s{+}q{-}1$. И во всяком случае, при проверке нового сдвига s' мы можем не рассматривать первые k символов образца: из формулы (34.5) мы знаем, что они заведомо совпадают с соответствующими символами в тексте.

Чтобы найти s', нам не нужно ничего знать о тексте T: достаточно знания образца P и числа q. Именно, $T[s'+1..s'+k]$ — суффикс строки P_q. Поэтому число k в формуле (34.5) — это наибольшее число $k < q$, для которого P_k является суффиксом P_q. Практически удобно хранить информацию именно об этом числе k — количестве символов, заведомо совпадающих при проверке нового сдвига s'. Само значение s' вычисляется по формуле $s' = s + (q-k)$.

Теперь дадим формальное определение. Префикс-функцией *(prefix function), ассоциированной со строкой $P[1..m]$, называется функция $\pi\colon \{1, 2, \ldots, m\} \to \{0, 1, \ldots, m-1\}$, определенная следующим образом:*

$$\pi[q] = \max\{\, k : k < q \text{ и } P_k \sqsupset P_q \,\}.$$

Рис. 34.9. Префикс-функция π. (а) Образец $P =$ ababaca расположен так, что первые 5 букв образца совпадают с буквами в тексте T (совпадающие буквы серые и соединены отрезками). (б) Исходя только из совпадения этих 5 букв, мы можем заключить, что сдвиг $s+1$ недопустим. Допустимость сдвига $s+2$ не противоречит тому, что мы к данному моменту знаем о тексте, и отбросить этот сдвиг заранее нельзя. (в) Информацию о том, какие сдвиги заведомо недопустимы, можно получить, исходя только из образца P. В нашем случае мы видим, что наибольший префикс строки P, являющийся суффиксом P_5 [и отличный от всей P_5], имеет длину 3. На языке префикс-функций это означает, что $\pi[5] = 3$. В общем случае: если при проверке сдвига s первые q символов образца совпали с соответствующими символами текста, то следующий сдвиг, который надо проверять, равен $s' = s + (q - \pi[q])$.

Иными словами, $\pi[q]$ — длина наибольшего префикса P, являющегося (собственным) суффиксом P_q. На рис. 34.10 (а) приведена префикс-функция для строки ababababca.

Алгоритм Кнута — Морриса — Пратта мы запишем в виде процедуры KMP-Matcher. *Как мы увидим,* KMP-Matcher *можно рассматривать как усовершенствование алгоритма* Finite-Automaton-Matcher. *Процедура* Compute-Prefix-Function, *вызываемая алгоритмом* KMP-Matcher, *вычисляет префиксную функцию π.*

```
KMP-Matcher(T,P)
n \gets length[T]
m \gets length[P]
\pi \gets Compute-Prefix-Function(P)
q \gets 0
for i \gets 1 to n
    do while q>0 and P[q+1] \ne T[i]
             do q \gets \pi[q]
       if P[q+1]=T[i]
          then q \gets q+1
       if q=m
          then print ``Образец входит со сдвигом ''i-m
               q \gets \pi[q]
```

```
Compute-Prefix-Function(P)
m \gets length[P]
\pi[1] \gets 0
k \gets 0
for q \gets 2 to m
    do while k>0 and P[k+1] \ne P[q]
             do k \gets \pi[k]
       if P[k+1]=P[q]
```

```
8           then k \gets k+1
9         \pi[q] \gets k
10 return \pi
```

Сначала мы проанализируем время работы этих процедур (в предположении их правильности), а затем докажем, что они работают правильно.

34.4.2 Время работы

Покажем, что время работы процедуры Compute-Prefix-Function *есть $O(m)$. Для этого воспользуемся методом потенциалов в амортизационном анализе (глава 18).*

Процедура Compute-Prefix-Function *выполняет $m-1$ итераций цикла в строках 4–9. Покажем, что учётную стоимость каждой из этих итераций можно считать равной $O(1)$. Для этого в качестве потенциала будем рассматривать значение k в момент входа в цикл. Поскольку начальное значение k равно нулю, а последующие равны $\pi[q] \geqslant 0$, реальная стоимость (время выполнения) всех итераций оценивается суммой учётных стоимостей, если считать учётной стоимостью сумму реальной стоимости и изменения потенциала.*

Реальная стоимость каждой итерации складывается из стоимости присваиваний в строке 6, присваивания в строке 8 и присваивания в строке 9. Поскольку $\pi[j] < j$ для всех j, при каждом присваивании в строке 6 потенциал уменьшается по крайней мере на 1, что компенсирует работу по выполнению действий в цикле **while**, *если умножить потенциал на достаточно большую константу. Поэтому вклад цикла* **while** *в учётную стоимость есть $O(1)$; присваивания же в строках 8 и 9 выполняются не более, чем по разу каждое, и увеличивают потенциал не более чем на две единицы. Поэтому учётная стоимость каждой итерации цикла* **for** *есть $O(1)$, а стоимость всей процедуры есть $O(m)$.*

Аналогичное рассуждение, в котором в качестве потенциала выбирается q, показывает, что время выполнения строк 5–12 процедуры KMP-Matcher *есть $O(n)$. Стало быть, время выполнения всей процедуры* KMP-Matcher *есть $O(m+n)$. Это — выигрыш по сравнению с алгоритмом* Finite-Automaton-Matcher, *который даже при экономном вычислении функции δ требует времени $O(m|\Sigma|+n)$.*

34.4.3 Префикс-функция вычисляется правильно

Начнем с важной леммы, показывающей, что, итерируя префикс-функцию, можно для данного q найти все такие k,

Рис. 34.10
Подпись:
Рис.34.10. К лемме 34.5 ($P =$ `ababababca`, $q = 8$). (а) Функция π, связанная со строкой P. Имеем $\pi[8] = 6$, $\pi[6] = 4$, $\pi[4] = 2$, $\pi[2] = 0$, так что $\pi^*[8] = \{\,8, 6, 4, 2, 0\,\}$. (б) Сдвигая строку P относительно самой себя слева направо, отмечаем моменты, когда префикс $P_k \sqsubset P$ является суффиксом строки P_8 (совпадающие символы выделены серым и соединены вертикальными отрезками; пунктирная линия нарисована справа от P_8). Это происходит при $k = 8, 6, 4, 2, 0$, что совпадает со множеством $\pi^*[8]$.

что P_k является суффиксом P_q. Именно, для данного q положим

$$\pi^*[q] = \{\,q, \pi[q], \pi^2[q], \pi^3[q], \ldots, \pi^t[q]\,\},$$

где $\pi^i[q]$ обозначает i-ую итерацию префикс-функции (то есть $\pi^0[q] = q$, а $\pi^i[q] = \pi[\pi^{i-1}[q]]$) и $\pi^t[q] = 0$ (такое t найдется, так как $\pi[j] < j$ для всех j; на этом месте итерации обрываются).

Лемма 34.5 (Лемма об итерациях префикс-функции) Пусть P — строка длины m с префикс-функцией π. Тогда для всех $q = 1, 2, \ldots, m$ имеем $\pi^*[q] = \{\,k : P_k \sqsupset P_q\,\}$.

Доказательство.

Покажем сначала, что

$$i \in \pi^*[q] \Longrightarrow P_i \sqsupset P_q. \tag{34.6}$$

В самом деле, $P_{\pi(i)} \sqsupset P_i$, так что каждый следующий член последовательности $P_q, P_{\pi[q]}, P_{\pi[\pi[q]]}, \ldots$ является суффиксом предыдущего (и, следовательно, всех предыдущих).

Покажем теперь, что и наоборот, $\{\,k : P_k \sqsupset P_q\,\} \subseteq \pi^*[q]$. Рассуждая от противного, обозначим через j наибольший элемент множества $\{\,k : P_k \sqsupset P_q\,\} \setminus \pi^*[q]$. Очевидно, $j < q$; раз j не лежит в последовательности $\pi^*[q]$, то j попадает между двумя её членами: $j' > j > \pi[j']$ для некоторого $j' \in \pi^*[q]$.

Обе строки P_j и $P_{j'}$ являются суффиксами P_q: первая — по выбору j, вторая — в силу соотношения (34.6). Стало быть, $P_j \sqsupset P_{j'}$ по лемме 34.1, и P_j является префиксом строки P, который является (собственным) суффиксом строки $P_{j'}$ и имеет длину больше $\pi[j']$, что противоречит определению функции π.

Рис. 34.10 иллюстрирует доказанную лемму.

Лемма 34.6

Пусть P — строка длины m с префикс-функцией π. Тогда $\pi[q] - 1 \in \pi^*[q-1]$ для всех $q = 1, 2, \ldots, m$. для которых $\pi[q] > 0$.

Доказательство.

Если $k = \pi[q] > 0$, то $P_k \sqsupset P_q$, откуда, отбрасывая последний символ, получаем, что $P_{k-1} \sqsupset P_{q-1}$. По лемме 34.5 это означает, что $\pi[q] - 1 \in \pi^*[q-1]$.

Для $q = 2, 3, \ldots, m$ определим множества E_{q-1} формулой

$$E_{q-1} = \{\, k : k \in \pi^*[q-1] \text{ и } P[k+1] = P[q] \,\}$$

(словами: E_{q-1} состоит из таких k, что P_k — суффикс P_{q-1}, и за ними идут одинаковые буквы $P[k+1]$ и $P[q]$, так что P_{k+1} — суффикс P_q).

Следствие 34.7

Пусть P — строка длины m с префикс-функцией π. Тогда для всех $q = 2, 3, \ldots, m$ имеем:

$$\pi[q] = \begin{cases} 0, & \text{если } E_{q-1} = \emptyset; \\ 1 + \max\{\, k \in E_{q-1} \,\}, & \text{если } E_{q-1} \neq \emptyset. \end{cases}$$

Доказательство.

Если $r = \pi[q] \geqslant 1$, то $P[r] = P[q]$; кроме того (лемма 34.6), $r - 1 \in \pi^*[q-1]$, так что $r - 1 \in E_{q-1}$. Отсюда ясно, что если E_{q-1} пусто, то $\pi[q] = 0$, и что если E_{q-1} непусто, то $\pi[q] \leqslant 1 + \max\{, k \in E_{q-1}\,\}$. Осталось показать, что если $k \in E_{q-1}$, то $\pi[q] \geqslant k + 1$ — но это ясно, поскольку в этом случае P_{k+1} есть суффикс P_q (как мы отмечали после определения E_{q-1}.

Теперь мы можем завершить доказательство правильности процедуры Compute-Prefix-Function. *всех q. Докажем индукцией по q следующее утверждение: при входе в цикл, начинающийся в строке 4, выполнено равенство $k = \pi[q-1]$. При $q = 2$ это обеспечивается присваиваниями в строках 2 и 3. Шаг индукции: пусть $k = \pi[q-1]$ при входе в цикл, тогда нам достаточно доказать, что при выходе из цикла будет $k = \pi[q]$. В самом деле, в строках 5–6 ищется наибольший элемент множества E_{q-1}; если это множество непусто (это равносильно тому, что $P[k+1] = P[q]$ при выходе из цикла в строках 5–6), то после присваивания в строке 9 число k оказывается равным $\pi[q]$ ввиду следствия 34.7. Если же это множество пусто (по выходе из цикла в строках 5–6 оказалось $k = 0$ и $P[k+1] \neq P[q]$), то оказывается $\pi[q] = 0$, что также верно (следствие 34.7).*

34.4.4 Алгоритм KMP правилен

Доказательство правильности процедуры KMP-Matcher *проходит по той же схеме, что и для процедуры* Compute-Prefix-Function. *Достаточно (индукцией по i) доказать такие утверждения:*

- *В момент исполнения строк 10–12 выполнено равенство $q = \sigma(T_i)$;*

- *Перед каждым исполнением тела цикла (строки 6–12) выполнено равенство*

$$q = \begin{cases} \sigma(T_{i-1}), & \text{если } \sigma(T_{i-1}) < m; \\ \pi[m], & \text{если } \sigma(T_{i-1}) = m. \end{cases}$$

(здесь $\sigma(T_j)$ — длина наибольшего префикса P, являющегося суффиксом T_j).

Заметим, что при $i = 1$ второе утверждение верно. Покажем теперь, что при каждом i из второго утверждения следует первое. В самом деле, пусть при некотором i второе утверждение верно. Лемма 34.5 показывает, что в строках 6–7 перебираются в убывающем порядке элементы множества $S = \{ k < m : P_k \sqsupset P_q \}$, причем перебор обрывается либо при нахождении наибольшего $k \in S$, для которого $P[k+1] = T[i]$, либо при $k = 0$ и $P[1] \neq T[i]$. В первом случае по выходе из цикла в строках 6–7 q равняется наибольшему k, для которого $P_k \sqsupset T_{i-1}$ и $P_{k+1} \sqsupset T_i$; ясно, что $k + 1 = \sigma(T_i)$, и после исполнения строки 9 значение $k + 1$ присваивается переменной q. Во втором случае по выходе из цикла в строках 6–7 имеем, очевидно, $\sigma(T_i) = 0$ и $q = 0$; в обоих случаях второе утверждение выполнено. Наконец, если второе утверждение верно для некоторого $i < n$, то, очевидно, первое утверждение верно для $i + 1$. Это завершает индукцию и доказательство.

Упражнения

34.4-1

Найдите префикс-функцию для строки ababbabbababbababbabb.

34.4-2

Укажите верхнюю оценку на размер множества $\pi^*[q]$ (как функцию от q) и покажите, что ваша оценка неулучшаема.

34.4-3

Как можно найти все вхождения образца P в текст T, зная префикс-функцию строки PT (конкатенация P и T)?

34.4-4

Покажите, что в алгоритме KMP-Matcher *можно в строке 7 (но не в строке 12) заменить π на π', где функция π' определена так:*

$$\pi'[q] = \begin{cases} 0, & \text{если } \pi[q] = 0; \\ \pi'[\pi[q]], & \text{если } \pi[q] \neq 0 \text{ и } P[\pi[q]+1] = P[q+1]; \\ \pi[q], & \text{если } \pi[q] \neq 0 \text{ и } P[\pi[q]+1] \neq P[q+1]. \end{cases}$$

Почему можно сказать, что такая модификация алгоритма KMP-Matcher *является его усовершенствованием?*

34.4-5

Укажите работающий за линейное время алгоритм, выясняющий, является ли данная строка T циклической перестановкой строки T' (например, строки `arc` *и* `car` *получаются одна из другой циклической перестановкой).*

34.4-6

Разработайте эффективный алгоритм, вычисляющий функцию перехода δ для конечного автомата, ищущего подстроку $P[1..m]$ в строке символов алфавита Σ. Ваш алгоритм должен работать за время $O(m|\Sigma|)$. (Указание. Докажите, что $\delta(q,a) = \delta(\pi[q],a)$, если $q = m$ или $P[q+1] \neq a$.)

34.5 Алгоритм Бойера — Мура

Если образец P длинный, а алфавит Σ достаточно велик, то наиболее эффективным алгоритмом поиска подстрок является, видимо, следующий алгоритм, изобретенный Бойером (Robert S. Boyer) и Муром (J. Strother Moore):

```
Boyer-Moore-Matcher(T,P,\Sigma)
1  n \gets length[T]
2  m \gets length[P]
3  \lambda \gets Compute-Last-Occurrence-Function(P,m,\Sigma)
4  \gamma \gets Compute-Good-Suffix-Function(P,m)
5  s \gets 0
6  while s \leqslant n-m
7        do j \gets m
8           while j>0 and P[j]=T[s+j]
9                 do j \gets j-1
10          if j=0
11             then print ‘‘Образец входит со сдвигом ’’s
12                  s \gets s+\gamma[0]
13             else s \gets s+\max(\gamma[j],j-\lambda[T[s+j]])
```

Если не обращать внимания на загадочные λ и γ, этот алгоритм очень похож на простейший алгоритм поиска подстрок. В самом деле, если мы закомментируем строки 3–4 и заменим строки 12–13 на

```
12                  s \gets s+1
13             else s \gets s+1,
```

то получится в точности простейший алгоритм разд. 34.1, с той единственной разницей, что сравнение $P[1..m]$ и $T[s+1..s+m]$ идет справа налево, а не слева направо.

Алгоритм Бойера — Мура вносит в простейший алгоритм со сравнением справа налево два усовершенствования, называемые ” *эвристикой стоп-символа" и "эвристикой безопасного суффикса" (см. рис. 34.11). Эти эвристики позволяют не рассматривать некоторые (на практике — весьма многие) значения сдвига s. Обе эвристики действуют независимо и используются одновременно. Если при проверке сдвига s обнаруживается, что подстрока $T[s+1..s+m]$ не совпадает с образцом, то каждая из эвристик указывает значение, на которое можно увеличить s, не опасаясь пропустить допустимый сдвиг (это $j-\lambda[T[s+j]]$ для эвристики стоп-символа и $\gamma[j]$ для эвристики безопасного суффикса); алгоритм Бойера — Мура выбирает из двух сдвигов больший.*

34.5.1 Эвристика стоп-символа

Стоп-символ, соответствующий данному сдвигу образца, — это первый справа символ в тексте, отличный от соответствующего символа в образце. Эвристика стоп-символа предлагает попробовать новое значение сдвига, исходя из того, где в образце встречается стоп-символ (если вообще встречается). В наиболее удачном случае стоп-символ выявляется при первом же сравнении (то есть $P[m] \neq T[s+m]$) и не встречается нигде в образце (представьте себе, что вы ищете подстроку $\mathtt{a}^m$ в строке $\mathtt{b}^n$). В этом случае сдвиг s можно сразу увеличить на m: любой меньший сдвиг заведомо не подойдет, так как стоп-символ в тексте окажется напротив какого-то символа из образца. Если этот наиболее удачный случай повторяется постоянно, то при поиске подстроки мы просмотрим всего лишь $1/m$ часть текста (вот как полезно сравнивать справа налево!).

В общем случае эвристика стоп-символа *(bad-character heuristic) работает так. Предположим, что при сравнении справа налево мы наткнулись на первое несовпадение: $P[j] \neq T[s+j]$, где $1 \leqslant j \leqslant m$. Пусть k — номер самого правого вхождения символа $T[s+j]$ в образец P (если этот символ вообще не появляется в образце, считаем k равным 0). Мы утверждаем, что можно увеличить s на $j-k$, не упустив ни одного допустимого сдвига. В самом деле, если $k=0$, то стоп-символ $T[s+j]$ вообще не встречается в образце P, так что можно сразу сдвинуть образец на $j-k=j$ позиций вправо (рис. 34.12 (а)); если $0<k<j$, то образец можно сдвинуть на $j-k$ позиций вправо, так как при меньших сдвигах стоп-символ в тексте не совпадёт с соответствующим символом образца (рис. 34.12 (б)). Наконец, если $k>j$, то эвристика предлагает сдвигать образец не вправо, а влево; алгоритм Бойера-Мура эту рекомендацию игнорирует, поскольку эвристика безопасного суффикса всегда предлагает ненулевой сдвиг вправо.*

Чтобы применять эвристику стоп-символа, полезно для ка-

Рис. 34.11, занимающий целую страницу. Переводы текстов на рисунке: bad character — стоп-символ, good suffix — безопасный суффикс.

Подпись:
Рис. 34.11. Эвристики Бойера — Мура (мы ищем в тексте подстроку `reminiscence`). (а) При сравнении сдвинутого на s образца с текстом (справа налево) выяснилось, что две крайние правые буквы совпадают (они образуют *”безопасный суффикс”* `ce`), а третья справа буква в образце — не такая, как в тексте (в тексте на этом месте стоит *”стоп-символ”* `i`: на нем сравнение строк обрывается). (б) Эвристика стоп-символа предлагает сдвинуть образец вправо на такое расстояние, чтобы стоп-символ в тексте оказался напротив крайнего правого вхождения этого символа в образец. В нашем случае это означает сдвиг на 4 позиции. Если стоп-символа в образце вообще нет, то образец надо полностью задвинуть за стоп-символ текста; если стоп-символ в образце встречается правее стоп-символа в тексте, то эвристика стоп-символа ничего полезного не предлагает (даёт отрицательный сдвиг, который будет проигнорирован алгоритмом). (в) Эвристика безопасного суффикса предлагает сдвинуть образец вправо настолько, чтобы ближайшее (если смотреть справа налево) вхождение безопасного суффикса в образец оказалось напротив безопасного суффикса в тексте. В нашем примере это означает сдвиг на 3 позиции. Алгоритм Бойера — Мура выбирает больший из двух рекомендуемых сдвигов (в нашем случае — сдвиг на 4).

ждого возможного стоп-символа $a \in \Sigma$ вычислить значение k. Это делается простой процедурой Compute-Last-Occurrence-Function *("найти последнее вхождение"), которая для каждого $a \in \Sigma$ вычисляет $\lambda[a]$ — номер крайнего правого вхождения a в P, или нуль, если a в P не входит. В этих обозначениях приращение сдвига, диктуемое эвристикой стоп-символа, есть $j - \lambda[T[s+j]]$, как и написано в строке 13 алгоритма* Boyer-Moore-Matcher.

```
Compute-Last-Occurrence-Function(P,m,\Sigma)
for (для) каждого символа a \in \Sigma
    do \lambda[a] \gets 0
for j \gets 1 to m
    do \lambda[P[j]] \gets j
return \lambda
```

Время работы процедуры Compute-Last-Occurrence-Function *есть $O(|\Sigma| + m)$.*

34.5.2 Эвристика безопасного суффикса

Если Q и R — строки, будем говорить, что они сравнимы *(обозначение: $Q \sim R$), если одна из них является суффиксом другой. Если выровнять две сравнимые строки по правому краю, то символы, расположенные один под другим, будут совпадать. Отношение $\sim$ симметрично: если $Q \sim R$, то и $R \sim Q$. Из леммы 34.1 следует, что*

$$\text{если } Q \sqsupset R \text{ и } S \sqsupset R, \text{ то } Q \sim S. \qquad (34.7)$$

Эвристика безопасного суффикса *(good-suffix heuristic) состоит в следующем: если $P[j] \neq T[s+j]$, где $j < m$ (и число j — наибольшее с таким свойством), то мы можем безбоязненно увеличить сдвиг на*

$$\gamma[j] = m - \max\{\, k : 0 \leqslant k < m \text{ и } P[j+1..m] \sim P_k \,\}.$$

Иными словами, $\gamma[j]$ — наименьшее расстояние, на которое мы можем сдвинуть образец без того, чтобы какой-то из символов, входящих в "безопасный суффикс" $T[s+j+1..s+m]$ оказался напротив не совпадающего с ним символа из образца. Поскольку строка $P[j+1..m]$ заведомо сравнима с пустой строкой P_0, число $\gamma[j]$ корректно определено для всех j. Стоит также заметить, что $\gamma[j] > 0$ для всех j, так что на каждом шаге алгоритма Бойера — Мура образец будет сдвигаться вправо хотя бы на одну позицию. Мы будем называть γ функцией безопасного суффикса *(good-suffix function), ассоциированной со строкой P.*

Посмотрим, как можно вычислить функцию безопасного суффикса γ. Для начала заметим, что $P_{\pi[m]} \sqsupset P$, откуда, в

Рис. 34.12, занимающий целую страницу. Перевод надписи в рисунке: bad character — стоп-символ.

Подпись:
Рис. 34.12. Эвристика стоп-символа: три случая. (а) Стоп-символ в образце не встречается, так что образец можно сдвинуть на $j = 11$ позиций вправо, оставив стоп-символ позади. (б) Крайнее правое вхождение стоп-символа в образец — в позиции $k < j$. Образец можно сдвинуть вправо на $j - k$ — так, чтобы стоп-символы в тексте и образце оказались друг под другом (в примере $j = 10$, $k = 6$, стоп-символ есть `i`, сдвиг на $10 - 6 = 4$). (в) Стоп-символ встречается в образце в позиции $k > j$ (в примере стоп-символ есть `e`, $j = 10$, $k = 12$). Эвристика предлагает сдвиг влево, но алгоритм это предложение игнорирует.

силу (34.7), имеем $P[j+1..m] \sim P_{\pi[m]}$ для любого j. Следовательно, максимум в правой части определения величины $\gamma[j]$ не меньше $\pi[m]$, так что $\gamma[j] \leqslant m - \pi[m]$ для всех j.

Стало быть, можно переписать наше определение γ так:

$$\gamma[j] = m - \max\{\, k : \pi[m] \leqslant k < m \text{ и } P[j+1..m] \sim P_k \,\}.$$

Условие $P[j+1..m] \sim P_k$ может выполняться в двух случаях: либо когда $P[j+1..m] \sqsupset P_k$, либо когда $P_k \sqsupset P[j+1..m]$. Во втором случае, однако, имеем $P_k \sqsupset P_m$, откуда $k \leqslant \pi[m]$ и потому $k = \pi[m]$. Поэтому определение γ можно переписать еще и так:

$$\gamma[j] = m - \max\left(\{\pi[m]\} \cup \{\, k : \pi[m] < k < m \text{ и } P[j+1..m] \sqsupset P_k \,\}\right)$$

Второе из этих множеств может оказаться пустым. В самом деле, мы ищем префикс P_k образца P, в котором $P[j+1..m]$ является суффиксом; другими словами, мы ищем в образце участок равный его суффиксу $P[j+1..m]$ и расположенный левее ($k < m$)

Нам нужно найти самый правый из таких участков (числа k, из которых берётся максимальное — это правые границы таких участков). Для этого полезно рассмотреть строку P', являющуюся обращением строки P и соответствующую ей префикс-функцию pi' (иными словами, $P'[i] = P[m+1-i]$ и $\pi'[t]$ — максимальное $u < t$, для которого $P'_u \sqsupset P'_t$).

Пусть участок X строки P заканчивается в позиции k (то есть является суффиксов строки P_k) и равен $P[j+1..m]$. Длины обоих равных участков равны $m - j$, и легко понять, что эти участки (в перевёрнутом виде) будут суффиксом и префиксом строки P'_l, где $l = (m-k) + (m-j)$ (здесь $(m-k)$ — расстояние от конца участка X до конца образца, $(m-j)$ — длина участка). Поэтому $\pi'[l] \geqslant m - j$. Докажем, что на самом деле имеет место равенство

$$\pi'[l] = m - j. \qquad (34.8)$$

В самом деле, если бы более длинный участок X', начинающийся в той же позиции, что и X, был бы суффиксом образца, то некоторый его суффикс совпадал бы с $P[j+1..m]$ и потому X не был бы самым правым участком строки P, совпадающим с $P[j+1..m]$,

Равенство (34.8) можно переписать как $j = m - \pi'[l]$; кроме того, из него следует, что $k = m - l + \pi'[l]$. поэтому можно окончательно переписать нашу формулу для $\gamma[j]$ так:

$$\begin{aligned}\gamma[j] &= m - \max(\{\pi[m]\} \cup \{\, m - l + \pi'[l] : 1 \leqslant< l \leqslant m \text{ и } j = m - \pi'[l] \,\})\\ &= \min(\{\, m - \pi[m] \,\} \cup \{\, l - \pi'[l] : 1 \leqslant l \leqslant m \text{ и } j = m - \pi'[l] \,\}) \qquad (34.9)\end{aligned}$$

(опять-таки, второе множество может оказаться пустым).

Теперь можно выписать псевдокод для функции, вычисляющей γ:

```
Compute-Good-Suffix-Function(P,m)
\pi \gets Compute-Prefix-Function(P,m)
P' \gets обращение строки P
\pi' \gets Compute-Prefix-Function(P',m)
for j \gets 0 to m
    do \gamma[j] \gets m-\pi[m]
for l \gets 1 to m
    do j \gets m - \pi'[l]
       if \gamma[j] > l - \pi'[l]
          then \gamma[j] \gets l - \pi'[l]
return \gamma
```

Процедура Compute-Good-Suffix-Function *проводит вычисления в точности по формуле (34.9). Её время работы есть* $O(m)$.

Время работы алгоритма Бойера — Мура в худшем случае есть $O((n-m+1)m+|\Sigma|)$, *поскольку на исполнение* Compute-Last-Occurrence-Function *уходит время* $O(m+|\Sigma|)$, *на* Compute-Good-Suffix-Function *уходит* $O(m)$, *и в худшем случае алгоритм Бойера — Мура (как и алгоритм Рабина — Карпа) потратит время* $O(m)$ *на проверку каждого априори возможного сдвига. На практике, однако, именно алгоритм Бойера-Мура часто оказывается наиболее эффективным.*

34.5.3 Упражнения

34.5-1

Вычислите функции λ *и* γ *для строки* $P =$ 0101101201.

34.5-2

Покажите на примерах, что эвристики стоп-символа и эвристика безопасного суффикса, действуя вместе, могут дать большой выигрыш по сравнению с использованием только эвристики безопасного суффикса.

*34-5.3**

На практике вместо функции γ *часто используют усовершенствованную функцию* γ', *определенную так:*

$$\gamma'[j] = m - \max\{\, k : 0 \leqslant k < m,\ P[j+1..m] \sim P_k \\ \text{и } (k-m+j>0 \Rightarrow P[j] \neq P[k-m+j])\,\}$$

(иными словами, мы дополнительно требуем, чтобы при новом сдвиге напротив стоп-символа в тексте стоял не тот же заведомо негодный символ образца, что при предыдущем). Как эффективно вычислять функцию γ'?

Задачи

34-1 Алгоритм поиска, учитывающий повторения в образце

Через y^i будем далее обозначать i-кратную конкатенацию строки y с собой (например, $(\mathrm{ab})^3 = \mathrm{ababab}$). Будем говорить, что строка $x \in \Sigma^*$ имеет коэффициент повторения *(has repetition factor) r, если $x = y^r$, где $r > 0$ и $y \in \Sigma^*$. Через $\rho(x)$ обозначим наибольший из коэффициентов повторения строки x.*

(*а*) *Разработайте эффективный алгоритм, находящий $\rho(P_i)$ для данной строки P и для всех $i = 1, 2, \ldots, m$. Оцените время работы вашего алгоритма.*

(*б*) *Для строки $P[1..m]$ положим $\rho^*(P) = \max_{1 \leqslant i \leqslant m} \rho(P_i)$. Пусть $|\Sigma| = 2$ и строка P выбирается случайным образом. Покажите, что математическое ожидание $\rho^*(P)$ ограничено сверху константой, не зависящей от m.*

(*в*) *Покажите, что приведенная ниже программа находит все вхождения образца P в текст T за время $O(\rho^*(P)n + m)$.*

```
Repetition-Matcher(P,T)
m \gets length[P]
n \gets length[T]
k \gets 1+\rho^*(P)
q \gets 0
s \gets 0
while s \leqslant n-m
     do if T[s+q+1]=P[q+1]
          then q \gets q+1
               if q=m
                 then print
                      ``Образец входит со сдвигом ''s
        if q=m или T[s+q+1]\ne P[q+1]
          then s \gets s+\max(1, \lceil q/k \rceil)
               q \gets 0
```

Этот алгоритм предложили Галил и Сейферас. Развивая эти идеи, они получили алгоритм для поиска подстрок, работающий за линейное время и использующий всего $O(1)$ памяти помимо требуемой для хранения P и T.

34-2 Параллельный алгоритм поиска

Обсудим, как можно использовать параллельный компьютер для поиска подстрок. Пусть для данного образца P у нас есть конечный автомат M, предназначенный для поиска P в тексте. Пусть Q — множество его состояний, а φ — функция, которая ставит в соответствие каждому слову состояние автомата M

после чтения этого слова. Пусть $T[1..n]$ *— текст, в котором мы ищем образец* P*, тогда нам необходимо найти* $\varphi(T_i)$ *для всех* i*. Мы будем действовать как в разделе 30.1.2.*

Для всякой строки x *обозначим через* $\delta_x(q)$ *состояние автомата* M*, в которое он придёт, если поместить его в состояние* q *и подать на вход строку* x*. (Таким образом,* δ_x *есть функция из* Q *в* Q*.)*

(а) *Покажите, что* $\delta_y \circ \delta_x = \delta_{xy}$ *(в левой части стоит композиция отображений).*

(б) *Покажите, что операция* $\circ$ *ассоциативна.*

(в) *Покажите, что таблица значений функции* δ_{xy} *может быть вычислена на CREW PRAM за время* $O(1)$ *исходя из таблиц значений* δ_x *и* δ_y*. Оцените необходимое для этого число процессоров в зависимости от* $|Q|$*.*

(г) *Покажите, что* $\varphi(T_i) = \delta_{T_i}(q_0)$*, где* q_0 *— исходное состояние.*

(д) *Покажите, что на CREW PRAM все вхождения образца* P *в текст* T *длины* n *можно найти за время* $O(\lg n)$ *(считайте, что образец* P *задан в виде соответствующего конечного автомата).*

34.6 Замечания

Связь задачи о поиске подстрок с конечными автоматами обсуждается у Ахо, Хопкрофта и Ульмана [4]. Алгоритм Кнута — Морриса — Пратта был независимо изобретен Кнутом и Моррисом, с одной стороны, и Праттом — с другой; в результате они опубликовали совместную работу. Рабин и Карп описали свой алгоритм в [117]; Бойер и Мур — в [32]. Галил и Сейферас [78] предложили интересный детерминированный алгоритм для поиска подстрок, работающий за линейное время и использующий лишь $O(1)$ *памяти (помимо памяти для хранения образца и текста).*

35 Вычислительная геометрия

Вычислительная геометрия — это раздел информатики, изучающий алгоритмы решения геометрических задач. Такие задачи возникают в машинной графике, проектировании интегральных схем, технических устройств и др. Исходными данными в такого рода задаче могут быть множество точек, набор отрезков, многоугольник (заданный, например, списком своих вершин в порядке движения против часовой стрелки) и т.п. Результатом может быть либо ответ на какой то вопрос (типа "пересекаются ли эти прямые?"), либо какой-то геометрический объект (например, наименьший выпуклый многоугольник, содержащий заданные точки).

В этой главе мы будем рассматривать только планиметрические задачи и проиллюстрируем некоторые важные методы вычислительной геометрии на этом примере. В наших задачах исходными данными будет множество точек плоскости $\{p_i\}$ *, заданных своими координатами:* $p_i = (x_i, y_i)$*, где* $x_i, y_i \in \mathbb{R}$*. Например, n-угольник P можно задать последовательностью* $\langle p_0, p_1, p_2, \ldots, p_{n-1}\rangle$ *его вершин в порядке обхода.*

В разделе 35.1 мы разберём простейшие задачи об отрезках (в какую сторону мы поворачиваем, двигаясь по ломаной, составленной из двух отрезков? пересекаются ли два отрезка?) и эффективные методы их решения. В разделе 35.2 мы применим так называемый " метод движущейся прямой" и построим алгоритм, определяющий за время $O(n \lg n)$*, есть ли среди n отрезков хотя бы два пересекающихся. В разделе 35.3 изложены два алгоритма, использующие "вращающуюся прямую" для вычисления выпуклой оболочки множества n точек: просмотр Грэхема, требующий времени* $O(n \lg n)$*, и проход Джарвиса, время работы которого* $O(nh)$*, где h — число вершин выпуклой оболочки. В разделе 35.4 рассмотрен алгоритм, который методом "разделяй и властвуй" за время* $O(n \lg n)$ *находит пару ближайших точек среди n заданных точек плоскости.*

35.1 Свойства отрезков

Начнем с некоторых определений, связанных с отрезками. Выпуклой комбинацией *(convex combination) двух различных точек* $p_1 = (x_1, y_1)$ *и* $p_2 = (x_2, y_2)$ *будем называть любую точку* $p_3 = (x_3, y_3)$*, для которой* $x_3 = \alpha x_1 + (1 - \alpha)x_2$ *и* $y_3 = \alpha y_1 + (1 - \alpha)y_2$ *при некотором* $0 \leqslant \alpha \leqslant 1$*. Это же условие можно записать как* $p_3 = \alpha p_1 + (1 - \alpha)p_2$*. Заданная таким образом точка* p_3 *принадлежит отрезку соединяющему* p_1 *и* p_2 *(и может совпадать с одним из концов). Это свойство можно принять за определение отрезка, назвав* отрезком *(line segment),* $\overline{p_1p_2}$ *множество всех выпуклых комбинаций* p_1 *и* p_2*. Точки* p_1 *и* p_2 *называют* концами *(endpoints) отрезка. Если важен порядок концов,говорят об* ориентированном отрезке *(directed segment)* $\overrightarrow{p_1p_2}$*. Если* p_1 *совпадает с точкой* $(0, 0)$*, называемой* началом координат *(origin), то ориентированный отрезок* $\overrightarrow{p_1p_2}$ *называют* вектором *(vector)* p_2*.*

Нас интересуют такие вопросы:

1. Даны два ориентированных отрезка $\overrightarrow{p_0p_1}$ *и* $\overrightarrow{p_0p_2}$ *с общим началом* p_0*. В какую сторону (по часовой стрелке или против неё) надо повернуть отрезок* $\overrightarrow{p_0p_1}$ *вокруг* p_0*, чтобы он пошёл в направлении* $\overrightarrow{p_0p_2}$*? (Имеется в виду меньший из двух поворотов.)*

2. Даны ломаная $p_1p_2p_3$*, составленная из двух отрезков* $\overline{p_1p_2}$ *и* $\overline{p_2p_3}$*. Идя по ней от* p_1 *к* p_3*, в какую сторону мы поворачиваем у* p_2 *— налево или направо?*

3. Пересекаются ли отрезки $\overline{p_1p_2}$ *и* $\overline{p_3p_4}$*?*

Мы сможем ответить на каждый из этих вопросов за время $O(1)$*, что, впрочем, не удивительно, поскольку исходными данными в каждом случае является фиксированное число точек. Более важно то, что наши методы не будут использовать ни деления, ни тригонометрических функций - операций, которые сложнее и более чувствительны к ошибкам округления, чем используемые нами сложение, вычитание, умножение и сравнение. Например, если (отвечая на вопрос номер 3) действовать наиболее очевидным способом и искать уравнения прямых, содержащих данные отрезки, в виде* $y = mx + b$ *(*m *— угловой коэффициент,* b *— ордината точки пересечения прямой с осью* y*), затем точку пересечения прямых, а затем проверять, принадлежит ли найденная точка обоим отрезкам, то понадобится деление, и для почти параллельных отрезков возможны сложности, вызванные округлением при делении на близкое к нулю число. (Мы укажем способ, позволяющий избежать деления.)*

Векторное произведение

Наше основное средство — понятие векторного произведения. Пусть даны вектора p_1 *и* p_2 *(рис. 35.1 (a)). Нас интересуют*

35.1 (a) Векторное произведение векторов p_1 и p_2 — это площадь параллелограмма (с учётом знака). (b) Вектора в светлой части плоскости получаются из вектора p поворотом по часовой стрелке, в тёмной — против.

только вектора, лежащие в одной плоскости, поэтому под векторным произведением *(cross product)* $p_1 \times p_2$ *можно понимать площадь параллелограмма (с учётом знака), образованного точками* $(0,0)$, p_1, p_2, $p_1+p_2 = (x_1+x_2, y_1+y_2)$. *Для вычислений более удобно определение векторного произведения как определителя матрицы*

$$p_1 \times p_2 = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1y_2 - x_2y_1 = -p_2 \times p_1$$

(Вообще говоря, векторное произведение определяется для векторов в трёхмерном пространстве как вектор, перпендикулярный p_1 *и* p_2, *образующий вместе с этими векторами правую тройку, длина которого равна площади параллелограмма, образованного точками* $(0,0)$, p_1, p_2, $p_1 + p_2$. *Но в этой главе нам достаточно определения векторного произведения на плоскости по формуле* $x_1y_2 - x_2y_1$.*)*

Если $p_1 \times p_2$ *положительно, то кратчайший поворот* p_2 *относительно* $(0,0)$, *совмещающий его с* p_1, *происходит по часовой стрелке, а если отрицательно, то — против. На рис. 35.1 (b) вектора в светлой части плоскости получаются из вектора* p *поворотом по часовой стрелке, в тёмной — против. Граница этих областей состоит из векторов, векторное произведение которых с* p *равно нулю. Эти векторы лежат на одной прямой с* p.

Напишем формулу, позволяющую определить, в какую сторону надо поворачивать $\overrightarrow{p_0p_1}$ *вокруг*p_0, *чтобы наложить его на* $\overrightarrow{p_0p_2}$ *(по часовой стрелке или против). Считая началом координат точку* p_0, *мы получаем два вектора* $p_1 - p_0 = (x_1 - x_0, y_1 - y_0)$ *и* $p_2 - p_0 = (x_2 - x_0, y_2 - y_0)$. *Их векторное произведение равно*

$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0).$$

Если оно положительно, то вращать надо против часовой стрелки ("в положительном направлении", как говорят), если отрицательно — то по часовой стрелке.

Направление поворота.

Перейдём ко второму вопросу: в какую сторону мы поворачиваем (в точке p_1*), двигаясь по ломаной* $\overline{p_0p_1p_2}$? *Легко избежать вычисления угла* $\angle p_0p_1p_2$, *вновь воспользовавшись векторным произведением. Как видно из рис. 35.2, нам надо выяснить, в каком направлении надо поворачивать вектор* $\overrightarrow{p_0p_1}$, *чтобы он стал направленным в сторону* $\overrightarrow{p_0p_2}$. *Для этого мы вычислим векторное произведение* $(p_2 - p_0) \times (p_1 - p_0)$. *Если оно отрицательно, то* $\overrightarrow{p_0p_2}$

35.2 Векторное произведение помогает определить, в какую сторону мы поворачиваем в точке p_1, идя по ломаной $p_0p_1p_2$. Для этого надо выяснить, в какой стороне находится вектор $\overrightarrow{p_0p_2}$ от вектора $\overrightarrow{p_0p_1}$. (a) Если против часовой стрелки, то мы поворачиваем налево. (b) Если по часовой стрелке, то — направо.

находится против часовой стрелки от $\overrightarrow{p_0p_1}$*, и в точке* p_0 *мы поворачиваем налево. Если векторное произведение положительно, мы поворачиваем направо. Наконец, если оно равно нулю, то мы либо идём прямо, либо поворачиваем кругом (точки* p_0, p_1 *и* p_2 *принадлежат одной прямой).*

Пересекаются ли отрезки?

Определять, пересекаются ли отрезки, мы будем в два этапа. Начальный тест *(quick rejection) состоит в следующем: если ограничивающие прямоугольники отрезков не имеют общих точек, то и сами отрезки не пересекаются. При этом* ограничивающим прямоугольником *(bounding box) геометрической фигуры мы будем называть наименьший из прямоугольников со сторонами, параллельными осям координат, которые содержат данную фигуру.*

Для отрезка $\overline{p_1p_2}$ *таковым будет прямоугольник* $(\hat{p}_1, \hat{p}_2)$ *с левым нижним углом* $\hat{p}_1 = (\hat{x}_1, \hat{y}_1)$ *и правым верхним углом* $\hat{p}_2 = (\hat{x}_2, \hat{y}_2)$ *, где* $\hat{x}_1 = min(x_1, x_2)$, $\hat{y}_1 = min(y_1, y_2)$, $\hat{x}_2 = max(x_1, x_2)$, $\hat{y}_2 = max(y_1, y_2)$*. Условие пересечения двух прямоугольников, заданных левыми нижними и правыми верхними углами, таково:* $(\hat{p}_1, \hat{p}_2)$ *и* $(\hat{p}_3, \hat{p}_4)$ *имеют общие точки тогда и только тогда, когда*

$$(\hat{x}_2 \geqslant \hat{x}_3) \wedge (\hat{x}_4 \geqslant \hat{x}_1) \wedge (\hat{y}_2 \geqslant \hat{y}_3) \wedge (\hat{y}_4 \geqslant \hat{y}_1)$$

(первые два условия соответствуют пересечению x*-проекций, вторые —* y*-проекциям).*

Если на первом этапе не удается установить, что отрезки не пересекаются, мы переходим ко второму этапу и проверяем, пересекается ли каждый из данных отрезков с прямой, содержащей другой отрезок. Отрезок пересекает *прямую (straddles a line), если его концы лежат по разные стороны от неё или если один из концов лежит на прямой. Можно проверить, что два отрезка пересекаются тогда и только тогда, когда пересекаются ограничивающие их прямоугольники (начальный тест) и, кроме того, каждый из отрезков пересекается с прямой, содержащей другой отрезок.*

Последнее условие проверяется с помощью векторных произведений. Точки p_3 *и* p_4 *лежат по разные стороны от прямой* p_1p_2*, если векторы* $\overrightarrow{p_1p_3}$ *и* $\overrightarrow{p_1p_4}$ *имеют различную ориентацию относительно вектора* $\overrightarrow{p_1p_2}$ *(см. рис.35.3 (a), (b)), то есть если знаки векторных произведений* $(p_3 - p_1) \times (p_2 - p_1)$ *и* $(p_4 - p_1) \times (p_2 - p_1)$ *различны. Если одно из этих векторных произведений равно нулю,* ВНИМАНИЕ!!! Рисунок, требующий добавлений. В книге есть ошибка, и после её исправления рисунок, называвшийся (e), называется (f), а на его месте надо поместить новый (см. на полях оригинала) Пересекает ли отрезок $\overline{p_3p_4}$ прямую, содержащую отрезок $\overline{p_1p_2}$? (a) Если да, то векторные произведения $(p_3-p_1)\times(p_2-p_1)$ и $(p_4-p_1)\times(p_2-p_1)$ имеют разные знаки. (b) Если нет, то эти произведения имеют один и тот же знак. (c)–(d) Граничные случаи, когда одно из векторных произведений равно нулю (и отрезок пересекает прямую). (e) Одно из векторных произведений равно нулю, ограничивающие прямоугольники пересекаются, но отрезки всё-таки не пересекаются (отрезок $\overline{p_1p_2}$ не пересекает прямую p_3p_4) (f) Отрезки лежат на одной прямой, но не пересекаются. Каждый из отрезков пересекает прямую, содержащую другой. Но сами отрезки не пересекаются — здесь это не страшно, так как этот случай отброшен на первом этапе.

то одна из точек p_3 или p_4 принадлежит прямой p_1p_2. Два таких случая показаны на рисунках (рис. 35.3 (c),(d)). В обоих случаях отрезки пересекаются, однако возможен и другой случай: хотя отрезки проходят начальный тест и векторное произведение равно нулю, всё-таки пересечения нет (рис. 35.3 (e)) Поэтому условие "отрезок пересекает прямую" надо проверять для обоих отрезков. Заметим, что начальный тест нельзя опустить: на рис. 35.3 (f) каждый из отрезков пересекает прямую, содержащую другой (лежит на ней — мы считаем это пересечением), но сами отрезки не пересекаются.

Если один из отрезков, например, p_1p_2, равен нулю, то вопрос о том, пересекает ли отрезок $\overline{p_3p_4}$ прямую p_1p_2, лишён смысла (прямых много). Тем не менее можно вычислить векторные произведения $(p_3-p_1)\times(p_2-p_1)$ и $(p_4-p_1)\times(p_2-p_1)$, которые в данном случае равны нулю. Наш критерий пересечения применим и в этом случае, если условно принять, что $\overline{p_3p_4}$ пересекает прямую p_1p_2.

Итак, критерий таков: отрезки p_1p_2 и p_3p_4 пересекаются тогда и только тогда, когда одновременно выполнены три условия:

(a) пересекаются ограничивающие их прямоугольники;

(b)

$$[(p_3-p_1)\times(p_2-p_1)]\cdot[(p_4-p_1)\times(p_2-p_1)]\leqslant 0$$

(c)

$$[(p_1-p_3)\times(p_4-p_3)]\cdot[(p_2-p_3)\times(p_4-p_3)]\leqslant 0$$

Другие применения векторного произведения.

Векторное произведение понадобится нам ещё не раз. В разделе 35.3 с его помощью мы будем сортировать точки по полярному углу относительно данного центра (см. упр. 35.1-2). В разделе 35.2 при построении красно-чёрного дерева отрезков, упорядоченных по месту их пересечения с данной вертикалью, мы будем ис-

пользовать формулу с векторным произведением, чтобы определить, какой из двух отрезков пересекает эту вертикаль в более высокой точке.

Упражнения

35.1-1

Докажите, что при положительном $p_1 \times p_2$ *вектор* p_1 *находится по часовой стрелке от вектора* p_2*, а при отрицательном — против. Оба вектора выходят из начала координат* $(0, 0)$*, бер/тся направление поворота в ближайшую сторону.*

35.1-2

Напишите алгоритм сортировки, располагающий точки $\langle p_1, p_2, \ldots, p_n\rangle$*. в порядке обхода против часовой стрелки, если смотреть из заданного полюса* p_0*. (Начать можно с любой точки.) Алгоритм должен использовать векторное произведение для сравнения углов и выполняться за время* $O(n \lg n)$*.*

35.1-3

Как за время $O(n^2 \ln n)$ *определить, лежат ли какие-то три из данных* n *точек на одной прямой?*

35.1-4

Чтобы узнать, являются ли точки $\langle p_0, p_1, \ldots, p_{n-1}\rangle$ *вершинами выпуклого многоугольника (подробнее о многоугольниках см. в разделе 16.4), перечисленными в порядке обхода многоугольника, профессор предлагает такой метод: проверить, что множество* $\angle p_i p_{i+1} p_{i+2} : i = 0, 1, \ldots, n-1$*, где* $i + 1$ *и* $i + 2$ *вычисляются по модулю* n*, не содержит одновременно правых и левых поворотов. Покажите, что этот способ не всегда дают правильный ответ, хотя и выполняется за линейное время. Как изменить его, чтобы получить правильный ответ (также за линейное время)?*

35.1-5

Правый горизонтальный луч *(right horizontal ray) точки* $p_0 = (x_0, y_0)$ *определяется как множество* $\{p_i = (x_i, y_i) : x - i \geqslant x_0 y_i = y_0\}$*, то есть как луч, выходящий из* p_0 *вправо параллельно оси абсцисс. Как за время* $O(1)$ *определить, пересекаются ли горизонтальный луч точки* p_0 *и отрезок* $\overline{p_1 p_2}$*, сведя задачу к исследованию пересечения двух отрезков?*

35.1-6

Чтобы узнать, лежит ли точка p_0 *внутри простого (не обязательно выпуклого) многоугольника* P*, можно рассмотреть какой-нибудь луч, выходящий из точки* p_0*, и посчитать, сколько раз он пересекает границу многоугольника: она будет внутренней, если число пересечений нечётно. Используя эту идею, напишите алгоритм, определяющий за время* $O(n)$*, является ли данная точка* p_0 *внутренней для простого* n*-угольника* P*. (Указание. Используйте упражнение 35.1-5. Особого рассмотрения требуют случаи, когда точка* p_0 *лежит на границе многоугольника, когда луч проходит через одну из его вершин или содержит участок стороны.)*

35.1-7

Покажите, как за время $\Theta(n)$ вычислить площадь простого (не обязательно выпуклого) n-угольника, заданного перечислением вершин в порядке обхода.

35.2 Есть ли пересекающиеся отрезки?

В этом разделе мы рассмотрим алгоритм, который определяет, есть ли среди n данных отрезков два пересекающихся. При этом используется метод "движущейся прямой", часто встречающийся в вычислительной геометрии. Другие применения этого метода указаны в упражнениях в конце раздела.

Алгоритм выполняется за время $O(n \ln n)$, где n — число данных отрезков. При этом проверяется только наличие или отсутствие пересечений; сами пересечения не находятся. (Как показывает упражнение 35.2-1, их отыскание может потребовать времени $\Omega(n^2)$.)

Метод движущейся прямой *(sweeping line method) состоит в том, что воображаемая вертикальная прямая движется слева направо мимо рассматриваемых геометрических объектов. Идея состоит в том, что от двумерного пространства мы переходим к произведению пространство-время (оба одномерны), считая ось абсцисс осью времени, движущуюся прямую — мгновенным срезом ситуации. При этом линии на плоскости становятся движущимися по прямой точками, и их можно упорядочивать. (Обычно они хранятся с помощью динамической структуры данных.) В задаче о пересечениях отрезков алгоритм просматривает концы всех отрезков слева направо и проверяет, нет ли пересечения на очередном участке оси абсцисс.*

Наш алгоритм использует два упрощающих предположения. Мы считаем, что среди рассматриваемых отрезков нет вертикальных и что никакие три отрезка не походят через одну и ту же точку (упр. 35.2-8 требует построить алгоритм, который годится во всех случаях.) Следует отметить, что возня с разнообразными граничными случаями — один из основных источников хлопот и ошибок при программировании задач вычислительной геометрии.

Отношения порядка на отрезках

Мы предположили, что среди рассматриваемых отрезков нет вертикальных. Поэтому движущаяся вертикальная прямая в любой момент пересекает каждый из них максимум в одной точке. Мы упорядочиваем отрезки (те, которые её пересекают) по ординате точки пересечения.

Более точно, два отрезка s_1 и s_2 сравнимы относительно x

Упорядочение отрезков относительно различных вертикальных прямых. (a) Для этой конфигурации $a >_r c, a >_t b, b >_t c, a >_t c$ и $b >_u c$; отрезок d не сравним ни с каким из отрезков на рисунке. (b) При проходе точки пересечения отношение порядка между отрезками e и f меняется на противоположное: $e >_v f$, а $f >_w e$. Когда движущаяся прямая z пересекает серую область, отрезки e и f являются соседними с точки зрения соответствующего ей порядка

(comparable at x), если вертикальная прямая, с абсциссой x, пересекается с ними обоими. При этом s_1 выше *s_2 относительно x (s_1 is above s_2 at x, обозначение $s_1 >_x s_2$), если отрезки s_1 и s_2 сравнимы относительно x и точка пересечения s_1 с вертикальной прямой находится выше точки пересечения s_2 с этой же прямой. На рисунке 35.4 (a) мы видим, что $a >_r c$, $a >_t b$, $b >_t c$, $a >_t c$ и $b >_u c$, а отрезок d не сравним ни с одним из остальных.*

Для любого фиксированного x отношение "$>_x$ " является отношением порядка (см. раздел 5.2) на множестве отрезков, пересекающихся с вертикальной прямой, проходящей через x. При разных значениях x этот порядок (как и множество, на котором он определён) может быть различным. Отрезок попадает в это множество, когда вертикальная прямая проходит через его левый конец, и выбывает из него, когда она проходит через правый конец.

Что происходит, когда вертикальная прямая проходит через точку пересечения двух отрезков? Отношение порядка между пересекающимися отрезками в точке пересечения меняется на противоположное (рис. 35.4 (b)): прямые v и w находятся слева и справа от точки пересечения отрезков e и f, при этом $e >_v f$ и $f >_u e$.

Напомним, что по нашему предположению никакие три отрезка не проходят через одну и ту же точку, поэтому в окрестности точки пересечения пересекающиеся отрезки непосредственно следуют один за другим (с точки зрения указанного порядка), рис. 35.4 (b).

Движение прямой.

Используя метод движущейся прямой, мы храним информацию двух видов:

1. Состояние дел у прямой *(sweep-line status) задаётся упорядоченным множеством объектов, пересекаемых движущейся прямой в текущий момент.*

2. Расписание *(event-point schedule) представляет собой последовательность моментов времени, в которых состояние может измениться (перечисленных в порядке возрастания времени). Такие моменты времени мы будем называть* критическими точками

35.5 Исполнение алгоритма Any-Segment-Intersect. Положения движущейся прямой в критических точках показаны пунктиром; рядом выписаны пересекаемые отрезки (в порядке сверху — вниз) в момент сразу после прохода критической точки, Пересечение отрезков d и b обнаруживается после удаления отрезка c.

(event point), так что изменение состояния дел у прямой возможно только в критической точке.

Для некоторых алгоритмов (см., например, упражнение 35.2-7) критические точки определяется постепенно по ходу работы алгоритма. Однако мы будем рассматривать алгоритм, в котором расписание легко найти заранее. В частности, абсцисса конца любого отрезка является критической точкой. Упорядочим концы отрезков в порядке возрастания их абсцисс. Отрезок начинает влиять на состояние дел у прямой с момента, когда прямая проходит через его левый конец, и перестаёт с момента, когда прямая проходит через его правый конец.

Состояние дел у прямой можно хранить как упорядоченное множество отрезков, с которым выполняются следующие операции:

Insert(T, s) *: добавить отрезок* s *в* T*;*

Delete(T, s) *: удалить отрезок* s *из* T*;*

Above(T, s) *: указать отрезок, располагающийся непосредственно выше* s *в множестве* T*;*

Below(T, s) *: возвращает отрезок, располагающийся непосредственно ниже* s *в множестве* T*.*

(Говоря об отношении порядка в T*, мы описываем слова " выше" и "ниже" в соответствии с геометрическим смыслом этого отношения.)*

Мы можем хранить (линейно) упорядоченное множество из n *отрезков и выполнять любую из указанных операций за время* $O(\lg n)$*, используя красно-чёрное дерево, Заметим, что операции сравнения (какой из отрезков выше в данный момент) можно выполнить за время* $O(1)$*, используя векторные произведения (см. упр. 35.2-2).*

Проверка пересечений

Построим алгоритм, который по заданному множеству S*, состоящему из* n *отрезков, проверяет, есть ли среди них хотя бы два пересекающихся. Этот алгоритм использует красно-чёрное дерево для хранения текущего состояния дел у движущейся прямой.*

```
Any-Segment-Intersect(S)
T \gets \emptyset
сортируем концы отрезков в порядке возрастания аб-
сцисс (точки
   с равными абсциссами идут в порядке возрастания ординат);
```

```
   проверяем, нет ли совпадающих точек среди концов (если есть -
   возвращаем true)
3  \emph{for} (для) каждой точки $p$ из полученного списка
4      \emph{do} if $p$ --- левый конец некоторого отрезка $s$
5            \emph{then}  Insert (T, s )
6                \emph{if}  (Above (T, s) существует и пересекает s) или
                            (Below (T, s) существует и пересекает s)
7                              \emph{then return} true
8               \emph{if} $p$ правый конец некоторого отрезка $s$
9                   \emph{then  if} определены Above (T, s) и Below (T, s)
                                        и (Above (T, s) пересекает Below (T, s)
10                              \emph{then return} true
11                                 Delete(T, s)
12  \emph{return} false
```

На рис. 35.5 показано выполнение алгоритма. Изначально множество T пусто (строка 1). В строке 2 строится расписание ожидаемых событий, соответствующих концам отрезков (вообще-то критическими точками являются также моменты пересечения отрезков, но после обнаружения такого пересечения алгоритм заканчивает работу сразу же, так что эти моменты мы смело можем игнорировать.

Каждая итерация цикла for *в строках 3–11 обрабатывает одну критическую точку. Если она соответствуют левому концу некоторого отрезка s, то в строке 5 этот отрезок добавляется в упорядоченное множество, а в строках 6–7 проверяется, не пересекается ли он часом с одним из соседних отрезков (если да, то алгоритм кончает работу и даёт ответ* TRUE*). (Возможно, что критическая точка соответствует концам нескольких отрезков — в этом случае они добавляются один за другим.)*

Если обрабатываемая критическая точка является правым концом некоторого отрезка s, то этот отрезок удаляется из T (строка 11); предварительно проверяется, не пересекаются ли отрезки, которые разделял удаляемый отрезок и которые теперь становятся соседними (строки 9–10)

Так делается для всех концов всех отрезков; если при этом пересечение так и обнаруживается, то алгоритм возвращает FALSE *(строка 12).*

Правильность алгоритма

Теорема 35.1

Процедура ANY-SEGMENT-INTERSECT*(S) возвращает* TRUE *в том и только том случае, когда среди отрезков множества S есть пересекающиеся.*

Доказательство.

Заметим, что значение TRUE *возвращается лишь после того, как обнаружены два пересекающихся отрезка. Поэтому надо лишь проверить, что если в S есть пара пересекающихся отрезков, то она будет обнаружена.*

Предположим теперь, что алгоритм не возвращает значения TRUE, *а просматривает по очереди все концы отрезков, не находит пересечения и возвращает значение* FALSE. *Покажем, что в таком случае отрезки не пересекаются.*

Рассмотрим сначала случай, когда концы всех отрезков имеют различные абсциссы. Тогда на каждой итерации цикла рассматривается новое значение абсциссы. Докажем, что при этом остаются верными такие свойства:

(a) множество T соответствует положению прямой непосредственно справа от последней обработанной абсциссы;

(b) соседние в множестве T отрезки не пересекаются.

Вначале эти свойства выполнены (T пусто, движущаяся прямая не пересекает ни одного из отрезков, находясь слева от всех них). Проверим, что они остаются выполненными после прохода через один из концов (другими словами, после очередной итерации цикла).

Если очередная точка есть левый конец отрезка, то появляется новая точка пересечения с движущейся прямой. Начинающийся в ней отрезок не пересекается с соседними (в смысле текущего состояния множества T) отрезками, иначе это было бы обнаружено. Остальные пары соседних отрезков были соседними раньше, так что мы уже знаем, что они не пересекаются.

Раз соседние отрезки не пересекаются, то на участке до следующего значения абсциссы в списке относительный порядок сохраняется (точка пересечения может быть только у несоседних отрезков и только после конца того отрезка, который их разделял).

Пусть теперь очередная точка есть правый конец отрезка. Тогда из множества T удаляется элемент (как и полагается, если мы хотим, чтобы T соответствовало положению прямой справа от очередной точки). Появляется новая пара соседних отрезков — про которую алгоритм проверяет, что они не пересекаются. По тем же причинам, что и раньше, можно утверждать, что до следующего значения абсциссы относительный порядок сохраняется.

Итак, свойства (a) и (b) проверены, и заодно доказано, что никакие два отрезка не пересекаются. В самом деле, раз между рассматриваемыми абсциссами порядок сохраняется, между ними *пересечение невозможно; а при прохождении движущейся прямой через конец отрезка пересечение также невозможно, так как некоторые пересекающиеся отрезки были бы соседними слева или справа от прямой.*

Осталось разобраться, почему наше рассуждение применимо к случаю, когда несколько концов отрезков лежат на одной вертикали (но не совпадают — это проверяется при сортировке концов отрезков). Это можно объяснить так: представим себе, что движущаяся прямая чуть-чуть наклонена влево (против часовой стрелки). Тогда она будет пересекать концы отрезков в разные (хотя, возможно, очень близкие) моменты времени, и просматриваться они будут как раз в нужном порядке (точки с одинаковой абсциссой — в порядке возрастания ординат). Тем самым можно свести дело к уже разобранному случаю (проверяться на пересечение будут те же пары отрезков, что и с воображаемой наклонённой прямой).

Время работы.

Если множество S *состоит из* n *отрезков, то время работы алгоритма* Any-Segment-Intersect *есть* $O(n \lg n)$. *В самом деле, сортировку можно выполнить за такое время (сортировку слиянием или с помощью двоичной кучи), затем мы обрабатываем* $2n$ *концов отрезков, тратя на каждый время* $O(\lg n)$ *(таково время любой использованной нами операции с красно-чёрным деревом). Заметим, что для одной пары отрезков выяснение того, пересекаются они или нет, выполняется за время* $O(1)$.

Упражнения.

35.2-1

Показать, что n *отрезков могут иметь* $\Theta(n^2)$ *точек пересечения.*

35.2-2

Даны два непересекающихся отрезка a *и* b, *сравнимых относительно* x. *Используя векторное произведение, определить за время* $O(1)$, *выполнено ли соотношение* $a >_x b$.

35.2-3

Профессор предлагает изменить программу Any-Segment-Intersect *так: найдя пересекающиеся отрезки, алгоритм не кончает работу, а продолжает выполнять цикл* for. *Получившийся алгоритм профессор называет* Print-Intersecting-Segments *и утверждает, что такой алгоритм напечатает все пересечения отрезков множества в порядке возрастания их абсцисс. Покажите, что профессор дважды неправ: во-первых, алгоритм может напечатать не все отрезки; во-вторых, первая напечатанная им точка может не быть самой левой точкой пересечения.*

35.2-4

Постройте алгоритм, который за время $O(n \lg n)$ *определяет,* *имеет ли данная замкнутая n-звенная ломаная самопересечения.*

35.2-5

Известно, что два несамопересекающиеся ломаные имеют в сумме n вершин. Постройте алгоритм, который за время $O(n \lg n)$ определяет, пересекаются ли они друг с другом.

35.2-6

Даны n кругов, заданных своими центрами и радиусами. Как определить за время $O(n \lg n)$, есть ли среди них два пересекающихся (то есть имеющих общую точку)?

35.2-7

Для набора из n отрезков имеется k точек пересечения. За время $O((n+k) \lg n)$ найти все точки пересечения.

35.2-8

Показать, как модифицировать алгоритм ANY-SEGMENT-INTERSECT*, чтобы он правильно работал для любого набора отрезков (в том числе и вертикальных; в некоторых точка может пересекаться более двух отрезков).*

35.3 Построение выпуклой оболочки

Выпуклой оболочкой *(convex hull) конечного множества точек Q называется наименьший выпуклый многоугольник, содержащий все точки из Q (некоторые из точек могут быть внутри многоугольника, некоторые — на его сторонах, а некоторые будут его вершинами). Выпуклую оболочку множества Q мы будем обозначать $CH(Q)$. Наглядно можно представлять себе дело так: в точках Q вбиты гвозди, на которые натянута резинка, охватывающая их все — эта резинка и будет выпуклой оболочкой множества гвоздей (рис. 35.6)*

В данном разделе мы рассмотрим два алгоритма отыскания выпуклой оболочки множества из n точек; результатом их работы будет список вершин выпуклой оболочки (в порядке обхода против часовой стрелки). Первый из алгоритмов (просмотр Грэхема) требует времени $O(n \lg n)$. Второй (проход Джарвиса) выполняется за время $O(nh)$, где h — число вершин выпуклой оболочки. Как подсказывает рис. 35.6, каждая вершина многоугольника $CH(Q)$ является одной из точек множества Q. Таким образом, для отыскания $CH(Q)$ алгоритм должен решить, какие вершины из Q оставить, а какие — отбросить.

Оба рассматриваемых нами алгоритма (просмотром Грэхема и проход Джарвиса) используют метод "вращающегося луча"

35.6 Выпуклая оболочка $CH(Q)$ множества Q (серый многоугольник) *(rotational sweep): выбирается полярная системы координат, и точки обрабатываются в порядке возрастания полярных углов (как во вращающемся радаре). Но существуют и другие методы поиска выпуклой оболочки за время $O(n \lg n)$; вот некоторые возможности.*

Метод добавления точек *(incremental method) рассматривает точки слева направо, располагая из в последовательность $(p_1, p_2, ..., p_n)$ в порядке возрастания абсцисс. На i-ом шаге строится выпуклая оболочка $CH(p_1, p_2, ..., p_i)$ первых i точек (к уже известной оболочке первых $i-1$ точек добавляется точка p_i). Эту идею можно превратить в алгоритм, находящий выпуклую оболочку за время $O(n \lg n)$ (упр. 35.3-6).*

Метод разделяй и властвуй *(divide-and-conquer method) за время $\Theta(n)$ делит множество из n точек вертикальной прямой на два примерно равных подмножества, затем рекурсивно ищет выпуклую оболочку каждого из них, а затем объединяет две эти оболочки за время $O(n)$.*

Метод стрижки и поиска *(prune-and-search method) напоминает алгоритм поиска медианы в упорядоченном множестве за линейное время (раздел 10.3). Он находит "верхнюю цепочку" выпуклой оболочки, повторно отбрасывая некоторую фиксированную долю вершин, пока не останутся только вершины из верхней цепочки. Аналогичным образом ищется нижняя цепочка. Асимптотически этот метод — самый быстрый из известных: если выпуклая оболочка содержит h вершин, то требуется время $O(n \lg h)$.*

Вычисление выпуклой оболочки важно не только само по себе. но и как промежуточный этап для многих задач вычислительной геометрии. Рассмотрим, например, двумерную задачу о наиболее удалённых точках *(farthest-pair problem): дано множество из n точек на плоскости; нужно выбрать пару максимально удалённых друг от друга точек. Можно доказать, что эти точки будут вершинами выпуклой оболочки (упр. 35.3-3); для выпуклого n-угольника можно найти две наиболее удалённые вершины за время $O(n)$ (идея: зажимаем его между двумя параллельными прямыми и вращаем). Тем самым можно найти две наиболее удалённые (из n) точек за время $O(n \lg n)$.*

Просмотр Грэхема

Просмотр Грэхема использует стек S, в котором хранятся точки. являющиеся кандидатами в выпуклую оболочку. Каждая точка исходного множества Q в некоторый момент помещается в стек; если она не является вершиной выпуклой оболочки $CH(Q)$, то через некоторое время точка будет удалена из стека, а если является — то останется. В момент окончания работы алгоритма в стеке S находятся в точности все вершины выпуклой оболочки $CH(Q)$ в порядке обхода против часовой стрелки.

Исходными данными для Graham-Scan *является множество*

35.7

Работа процедуры textscGraham-Scan для множества Q (рис. 35.6). Для каждого из шагов текущее содержимое стека S показано серым. (a) Точки $\langle p_1, p_2, ...p_m \rangle$ отсортированы по полярному углу относительно p_0; в стеке — первые три точки (p_0, p_1, p_2). (b) — (k) Стек S после очередной итерации цикла *for* (строки 7 — 10). Пунктирными линиями показаны повороты, которые не являются левыми; в этом случае вершина угла поворота удаляется из стека. Например, на этапе (h) из стека сначала удаляется точка p_8 (правый поворот $p_7p_8p_9$), а затем точка p_7 (правый поворот $p_6p_7p_9$). (l) Результат работы алгоритма — выпуклая оболочка — совпадает с приведённой на рис. 35.6.

Q из (не менее чем трёх) точекг. Процедура используем функцию Top(S)*, которая возвращает точку, находящуюся в вершине стека S (не меняя содержимого стека), а также функцию* Next-To-Top(S)*, которая возвращает следующий за вершиной элемент стека, также не меняя содержимого. Мы докажем, что по окончании работы в стеке (от дна к вершине) идут как раз вершины $CH(Q)$ (в порядке обхода против часовой стрелки).*

```
Graham-Scan(Q)
1 пусть $p_0$ --- точка из множества $Q$ с наимень-
шей ординатой
     (если таковых несколько --- самая левая)
2 пусть $\langle p_1, p_2, ... p_m \rangle$ --- осталь-
ные точки
   множества $Q$, отсортированные в порядке возрастания
   полярного угла (против часовой стрелки) относительно точки
   $p_0$ Если есть несколько точек с одинаковым поляр-
ным углом,
   то оставляем только самую удалённую от $p_0$.
3  сделать стек $S$ пустым
4  $Push(S,p_0)$
5  $Push(S,p_1)$
6  $Push(S,p_2)$
7  \emph{for} $i\leftarrow 3 \emph{to} m$
8          \emph{do while} при движении по ломаной
               $Next-To-Top(S) \rightarrow Top(S) \rightarrow p_i$
               мы двигаемся прямо или направо
9                     \emph{do}  $Pop(S)$
10         $Push(S,p_i)$
11  \emph{return} $S$
```

На рисунке 35.7 показаны шаги выполнения алгоритма Graham-Scan. *В строке 1 мы находим точку p_0; все остальные точки множества Q находятся выше неё (или на одном уровне,* рисунок совпадает с 35.8 (b), только нет точки p_i, пунктирных линий $p_0p_ip_j$ и серой области

35.8 Если точки $p_1, \dots, p_k, p_j$ идут в порядке возрастания полярных углов (относительно точки p_0), которые меняются в промежутке от $[0, \pi)$, и при движении по ломаной $p_0p_1 \dots p_kp_j$ мы в каждой вершине поворачиваем налево, то выпуклой оболочкой точек $p_0, p_1, \dots, p_k, p_l$ будет многоугольник, вершинами которого будут все эти точки, а границей будет ломаная $p_0p_1 \dots p_kp_jp_0$

но правее), и потому она заведомо ю входит в $CH(Q)$. В строке 2 оставшиеся точки множества Q упорядочиваются полярному углу относительно точки p_0. Каждое сравнение можно выполнить за время $O(1)$ с помощью произведений (упр. 35.1-2); из точек с одинаковым углом остаётся самая дальняя (остальные заведомо не принадлежат выпуклой оболочке). Полярный углы точек множества Q находятся в пределах промежутка $[0, \pi)$. Заметим, что точки p_1 и p_m являются вершинами $CH(Q)$ (упр. 35.3-1).

Затем (строки 3–6) мы помещаем в стек три первые точки p_0, p_1, p_2, после чего можем утверждать, что стек содержит выпуклую оболочку множества $p_0, p_1, \dots, p_k$ при $k = 2$. Это свойство будет поддерживаться и в дальнейшем, а k будет расти, пока не станет равным m. При движении от дна стека к его вершине мы всё время поворачиваем налево. Если просто так добавить очередную точку в стек, мы нарушим это свойство, поэтому предварительно мы выбрасываем несколько верхних вершин стека.

Теорема 35.2 (алгоритм Грэхема правилен)

После исполнения процедуры Graham-Scan *для множества точек Q, содержащего по меньшей мере три точки, в стеке S содержится выпуклая оболочка множества Q (в порядке обхода против часовой стрелки).*

Доказательство.

По-настоящему строгое доказательство выходит за рамки книги (строгое определение внутренности многоугольника — уже непростая задача), поэтому мы лишь укажем два наглядно очевидных факта, которые гарантируют правильность алгоритма; они показаны на рис. 35.8 и 35.9

ВНИМАНИЕ: ЗДЕСЬ НОВЫЕ РИСУНКИ!!!

Покажем теперь, что время работы алгоритма Graham-Scan *есть $O(n \lg n)$, где $n = |Q|$. Сортировка укладывается в эти границы, если использовать сортировку слиянием или с помощью кучи. Остальная часть алгоритма требует времени $O(n)$. Это не сразу очевидно, поскольку очередная итерация цикла* **for**, *помимо добавления одной новой точки, может потребовать удале-*

[рисунок совпадает с 35.8 (а), только добавлено обозначение p_1 для точки рядом с p_0]

35.9 Если точки $p_1, \ldots, p_k, p_j, p_i$ идут в порядке возрастания полярных углов (относительно точки p_0), которые меняются в промежутке от $[0, \pi)$, и при движении по двум соседним участкам $p_k p_j p_i$ ломаной $p_0 p_1 \ldots p_k p_j p_i$ мы поворачиваем вправо или идём прямо, то выпуклая оболочка точек $p_1, \ldots, p_k, p_j, p_i$ не изменится после удаления точки p_j (поскольку она лежит внутри треугольника $p_0 p_k p_i$).

ния нескольких (и, возможно, многих) старых.

Тем не менее, методы амортизационного анализа позволяют легко доказать, что общее число действий есть $O(n)$. В самом деле, любая точка p_i добавляется в стек S только один раз, а потому и удаляется не более одного раза. Тем самым общее время и на добавление, и на удаление есть $O(n)$. (Аналогичное рассуждение мы использовали в разделе 18.1 при анализе процедуры MULTIPOP.*)*

Проход Джарвиса

Проход Джарвиса вычисляет выпуклую оболочку множества точек Q при помощи "заворачивания" (package wrapping, gift wrapping). Алгоритм требует времени $O(nh)$, где h — число вершин выпуклой оболочки. Поэтому, для класса задач, где h есть $o(\lg n)$, проход Джарвиса асимптотически быстрее просмотра Грэхема.

Идея алгоритма такова: мы хотим обвязать наши гвозди верёвкой. Конец верёвки прикрепим к нижнему гвоздю p_0 (которая выбирается как и в предыдущем алгоритме), и пустим верёвку вправо. Затем будем поднимать правый конец верёвки, пока она не коснётся некоторого гвоздя p_1. Дальше мы вращаем верёвку уже относительно p_1, пока она не коснётся следуюёщего гвоздя p_2. Так продолжается, пока верёвка не дойдёт до исходной точки p_0.

Более формально, последовательность точек $H = \langle p_0, p_1, \ldots, p_h - 1\rangle$, являющихся вершинами $CH(Q)$, строится так. Мы начинаем с точки p_0. Следующая точка p_1 имеет наименьший (среди всех точек множества Q) полярный угол относительно p_0 (см. рис. 35.10) (Если таковых несколько, выбираем самую далёкую.) Затем мы, стоя в точке p_1, сравниваем полярные углы всех точек и выбираем точку p_2 с наименьшим полярным углом (относительно p_1), и так далее. В какой-то момент мы дойдём до верхней точки (точки с наибольшей ординатой), после чего процесс продолжится, только теперь надо отсчитывать полярные углы от луча, направленного влево, а не вправо. Тем самым мы построим сначала (до верхней точки) правую цепь *(right chain) выпуклой оболочки, а затем* левую цепь

(рис. 35.10)

На самом деле можно было бы обойтись без выделения правой и левой цепи. Для этого можно обычно хранить направление последней найденной стороны и выбирать точку с ближайшим (в положительную сторону) направлением луча. Впрочем, в этом случае несколько сложнее сравнивать углы (при использовании левой и правой цепи сравнение углов можно произвести с помощью приёмов раздела 35.1, не вычисляя их явно.

При естественной реализации проход Джарвиса выполняется за время $O(nh)$. Для каждой вершины h выпуклой оболочки $CH(Q)$ мы ищем вершину с минимальным полярным углом. Каждое сравнение углов выполняется за время $O(1)$ (при этом можно обойтись без тригонометрии, использовав приёмы разд. 35.1), так что поиск минимума требует времени $O(n)$ (разд. 10.1). Общее время, будет, таким образом, $O(nh)$.

Упражнения

35.3-1

Докажите, что точки p_1 и p_m в процедуре Graham-Scan *окажутся вершинами $CH(Q)$.*

35.3-2

Будем считать, что сложение, умножение и сравнение выполняются за время $O(1)$. Покажите, что если мы умеем находить выпуклую оболочку n точек (вершины которой требуется указать в порядке обхода против часовой стрелки) быстрее чем за $\Omega(n \lg n)$ шагов, то можно отсортировать n чисел быстрее чем за $\Omega(n \lg n)$ шагов.

35.3-3

В множестве точек Q возьмём две наиболее удалённые друг от друга точки. Покажите, что обе они являются вершинами выпуклой оболочки.

35-3.4

Пусть даны многоугольник P и точка q на его границе. Тенью *(shadow) точки q называется множество всех точек r, для которых что отрезок $\overline{qr}$ не выходит за пределы многоугольника P. Многоугольник P называется* звёздным *(star-shaped), если внутри него есть точка p, из которой видна вся его граница (другими словами, p принадлежит тени любой точки границы многоугольника P). Множество всех таких точек p называется* ядром *(kernel) многоугольника P (см. рис. 35.11). Даны вершины зв/здного n-угольника P, перечисленные в порядке обхода границы против часовой стрелки. Покажите, как можно найти $CH(P)$ за время $O(n)$.*

35.3-5

Задача об отыскании выпуклой оболочки множества Q в режиме *"on-line" (on-line convex-hull problem) состоит в следующем: нам дают n точек одну за другой; в каждый момент времени*

35.11
(К упр. 35.3-4)
(a) Звёздный многоугольник: из точки p видна любая точка его границы.
(b) Многоугольник, не являющийся звёздным. Слева закрашена тень точки q, справа — тень точки q'. Ядро пусто, так как эти области не пересекаются.

мы должны указать выпуклую оболочку всех точек, полученных к этому моменту. Можно применить проход Грэхема на каждом шаге заново, и тогда общее время работы алгоритма будет равно $O(n^2 \lg n)$. Придумайте алгоритм, требующий времени $O(n^2)$.

*35.3-6**

Используя просмотр точек слева направо, найдите выпуклую оболочку n точек за время $O(n \lg n)$.

35.4 Отыскание пары ближайших точек

Пусть теперь нам надо найти среди $n \leqslant 2$ точек множества Q пару ближайших друг к другу точек. Расстояние между точками понимается в обычном смысле: точки $p_1 = (x_1, y_1)$ и $p_2 = (x_2, y_2)$ находятся на расстоянии $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

Вообще говоря, две точки могут совпадать (тогда расстояние между ними равно 0). Такая задача может возникнуть, например, в системах контроля за транспортом: полезно знать, какие два транспортных средства ближе всего друг к другу (риск столкновения)

Если искать пару ближайших точек "в лоб", надо перебрать все $C_n^2 = \Theta(n^2)$ пары точек. В этом разделе мы с помощью метода "разделяй и властвуй" построим алгоритм, время работы которого описывается рекуррентным соотношением $T(n) = 2T(n/2) + O(n)$, т.е. равно $O(n \lg n)$.

Метод "разделяй и властвуй" для отыскания ближайших точек.

Входные данные каждого рекурсивного вызова алгоритма состоят из подмножество $P \subseteq Q$ и двух массивы X и Y. Каждый из массивов содержит точки подмножества p, но порядок в них разный: в массиве X точки расположены в порядке возрастания абсцисс, а в массиве Y — в порядке возрастания ординат. Заметим, что мы не можем позволить себе сортировать точки при каждом вызове, так как в этом случае получится соотношение (как минимум) $T(n) = 2T(n/2) + O(n \lg n)$, т.е. $T(n) = O(n \ln^2 n)$. (Мы увидим, как эту трудность можно обойти с помощью " предсортировки".

Итак, пусть мы получили P,X и Y. Если $|P| \leqslant 3$, перебираем все пары точек (максимум три) и сравниваем расстояние. Если же $|P| > 3$, мы поступаем так:

Разделяй (divide): Находим вертикальную прямую l, которая делит множество P на два подмножества P_L и P_R половинного размера ($|P_L| = \lceil |P|/2 \rceil$, $|P_R| = \lfloor |P|/2 \rfloor$; точки, лежащие на прямой l, как-то поделены между P_L и P_R). Массив X делим на массивы X_L и X_R, содержащие точки подмножеств P_L и P_R (сохраняя порядок); массив Y делится на массивы Y_L и Y_R аналогичным образом.

Властвуй (conquer): После деления P на P_L и P_R, выполняем два рекурсивных вызова и находим пару ближайших точек в множестве P_L (входные данные этого вызова — подмножество P_L и массивы X_L и Y_L), а также пару ближайших точек в множестве P_R (входные данные — P_R, X_R и Y_R). Обозначим расстояния между ближайшими точками в подмножеств P_L и P_R через δ_L и δ_R. Положим $\delta = \min(\delta_L, delta_R)$.

Соединяй (combine): Для всего множества P парой ближайших точек является либо одна из найденных пар точек (расстояние δ), либо некоторая "приграничная" пара точек, в которой одна точка принадлежит множеству P_L, а другая — P_R (если среди таких пар есть пара с расстоянием меньше δ). Очевидно, что точки такой приграничной пары отстоят от прямой l не более, чем на δ, т.е. находятся в симметричной относительно l вертикальной пограничной полосе шириной 2δ (рис. 35.12 (а)). Чтобы найти такую приграничную пару, если она существует, проделаем следующее.

1. Создадим массив $Y\prime$, поместив в него все точки из массива Y, которые попадают в пограничную полосу (с той или иной стороны от прямой l). (Порядок сохраняем: массив $Y\prime$ отсортирован по ординатам точек.)

2. Для каждой точки p массива Y' ищем такие точки массива Y', которые удалены от p не более, чем на δ. Как мы вскоре увидим, достаточно рассмотреть только 7 соседних (в порядке возрастания ординат) точек в массиве Y'. Мы вычисляем расстояние от p до каждой из этих 7 точек. Выполнив это для всех точек p из Y', находим $\delta\prime$ — расстояние между точками массива $Y\prime$, расположенными ближе всего друг к другу.

3. Если $\delta' < \delta$, то пара ближайших точек находится внутри вертикальной полосы и мы возвращаем эту пару точек и расстояние δ'. В противном случае мы возвращаем пару, найденную при одном из рекурсивных вызовов, и расстояние δ.

Сейчас мы докажем правильность этого алгоритма, а затем обсудим детали его реализации (необходимые, чтобы уложиться в $O(n \lg n)$ действий.

Правильность алгоритма

Нужно понять лишь, почему достаточно сравнивать каждую точку полосы лишь с семью следующими за ней (в порядке возрастания ординаты). Сейчас мы в этом убедимся.

Пусть (при некотором вызове) ближайшей парой точек является пара из точек $p_L \in P_L$ и $p_R \in P_R$ (рис. 35.12 (a)), и расстояние δ' между этими точками строго меньше δ. Точка p_L должна находиться слева от l на расстоянии не более δ (или на самой прямой); p_R — справа на расстоянии не большем δ (или на прямой). Кроме того, расстояние по вертикали между p_L и p_R также не превосходит δ. Следовательно (рис. 35.12 (a)), точки p_L и p_R можно поместить в прямоугольник размера $\delta \times 2\delta$, симметричной относительно прямой l. (Конечно, в него могут попасть и другие точки.)

Покажем теперь, что внутри этого прямоугольника может быть не более 8 точек из множества P. Рассмотрим его левую половину — квадрат $\delta \times \delta$. Все точки из P_L находятся на расстоянии не менее δ друг от друга, поэтому в этом квадрате может быть максимум 4 таких точки (в каждой его четвертинке может быть не более одной точки), см. рис. 35.12 (b). В правой половине прямоугольника (не считая границы) точек из P_L быть не может, так что всего в прямоугольнике не более 4 точек из P_L. По аналогичным причинам там не более 4 точек из P_R. Заметим, что случай 8 точек действительно возможен: четыре точки из P_L могут быть в углах левого квадрата, а четыре точки из P_R — в углах правого (две точки будут общими для P_L и P_R). а Теперь ясно, что что для каждой точки достаточно проверить 7 точек, непосредственно следующих за ней в массиве Y': если между двумя ближайшими точками есть ещё 7 промежуточных (в порядке возрастания ординат), то все 9 точек попадут внутрь прямоугольника, а этого быть не может, как мы видели.

Детали реализации и время работы алгоритма.

Наша цель — получить для времени работы алгоритма рекуррентное соотношение $T(n) = 2T(n/2) + O(n)$ (здесь $T(n)$ — время обработки n точек). Чтобы этого достичь, мы должны предполагать, что входные массивы X и Y отсортированы по абсциссе и ординате соответственно (тогда разделить множество на левую и правую половины можно за линейное время). Соответственно мы обязаны при рекурсивном вызове передавать множества P_L и P_R в отсортированном виде. Ясно, что это нетрудно сделать, проходя массив от начала к концу и раскладывая его элементы на две группы. В некотором смысле это действие обратно в процедуре MERGE, *использованной при сортировке слиянием (раздел 1.3.1).*

Осталось заметить, что начальная сортировка *(presorting) выполняется всего один раз и потому не входит в рекуррентное со-*

отношение — её время нужно просто добавить к времени работы рекурсивной процедуры. Время этой сортировки есть $O(n \lg n)$, так что общее время есть сумма двух слагаемых вида $O(n \lg n)$ и потому есть $O(n \lg n)$.

Упражнения

35.4-1

Профессор предлагает усовершенствовать алгоритм поиска пары ближайших точек и проверять только 5 точек массива Y'. Он говорит следующее: "Будем относить точки, принадлежащие прямой l, к множеству P_R. В этом случае на прямой l не будет совпадающих точек, которые относятся одновременно к двум множествам P_L и P_R. Поэтому внутри прямоугольника $\delta \times 2\delta$ будет содержаться не более 6 точек." Что неправильно в идее профессора?

35.4-2

Покажите, что (без изменения асимптотики времени работы алгоритма) можно подавать на вход алгоритма множество, не содержащее совпадающих точек. Покажите, что, если совпадающих точек в множестве нет, то для каждой точки массива Y' достаточно проверять 6 (а не 7) соседних точек. Объясните, почему нельзя обойтись 5 ближайшими точками.

35.4-3

*Определим L_m-*расстояние *(L_m-distance) между точками p_1 и p_2 плоскости формулой $(|x_1 - x_2|^m + |y_1 - y_2|^m)^{1/m}$. Евклидово расстояние соответствует значению $p = 2$. Измените приведённый в этом разделе алгоритм так, чтобы он искал ближайшие точки в смысле L_1-расстояния.*

35.4-4

Определим L_∞-расстояние между точками p_1 и p_2 как $\max(|x_1 - x_2|, |y_1 - y_2|)$. Как приспособить алгоритм поиска ближайших точек к случаю L_∞-расстояния?

35.5 Задачи

35-1 Выпуклые слои

Определим emphвыпуклые слои (convex layers) множества Q точек плоскости. Первый выпуклый слой Q содержит точки множества Q, которые являются вершинами $CH(Q)$. Выбросим эти эти точки и снова найдём выпуклую оболочку — её вершины образуют второй выпуклый слой. Выбросим и их; возьмём выпуклую оболочку остатка; её вершины образуют третий выпуклый слой и т.д.

a. Придумать способ за время $O(n_2)$ найти все выпуклые слои множества из n точек плоскости.

b. Покажите, что задачу сортировки n вещественных чисел можно свести к задаче отыскания выпуклых слоёв, и потому в любой модели, где сортировка требует времени $|Omega(n \lg n)$*, та же оценка верна и для задачи отыскания выпуклых слоёв.*

35-2

Максимум-слои

Будем говорить, что точка плоскости (x, y) мажорирует *(dominates) точку* (x', y')*, если* $x \geqslant x'$ *и* $y \geqslant y'$*. Пусть* Q *— множество точек плоскости. Если для точки из множества* Q *нет других точек из* Q*, мажорирующих её, то такую точку будем называть* максимальной *(maximal) в множестве* Q*. Таких точек может быть несколько; множество таких точек называют первым* максимум-слоем. *Удалив все эти точки, повторим процесс с оставшимся множеством точек, получим второй максимум-слой, и будет это повторять до тех пор, пока точки не кончатся.*

Предположим, что множество Q *имеет* k *непустых максимум слоев* $L_1, L_2, \ldots, L_k$*. Пусть* y_i *— ордината самой левой точки слоя* L_i *для* $i = 1, 2, \ldots k$*. Будем предполагать, что в* Q *нет двух точек с совпадающими абсциссами или ординатами.*

a. Докажите, что $y_1 > y_2 > \ldots > y_k$*.*

Пусть точка (x, y) *находится левее любой из точек* Q*, а её ордината* y *отлична от ординат всех точек множества* Q*. Пусть* $Q' = Q \cup \{(x, y)\}$*.*

b. Обозначим через j *наименьший из индексов, для которых* $y_j < y$*; если такого индекса нет* ($y < y_k$)*, положим* $j = k + 1$*. Покажите, что максимум-слои множества* $Q\prime$ *устроены так:*

Если $j \leqslant k$*, то все максимум-слои* Q' *совпадают с максимум-слоями* Q*, за исключением слоя* L_j*, в который добавилась точка* (x, y) *(и стала самой левой точкой нового* L_j*).*

Если $j = k + 1$*, то первые* k *максимум-слоёв множества* Q' *такие же, как у* Q*, но кроме них есть непустой* $(k + 1)$*-ый слой, состоящей из единственной точки* (x, y)*.*

c. Опишите алгоритм, вычисляющий все максимум-слои множества из n точек за время $O(n \lg n)$*. (Указание. Двигайте прямую справа налево.)*

d. Какие сложности возникнут, если среди точек есть точки с равными абсциссами или ординатами, и как можно поступить в таком случае?

35-3 Роботы и призраки.

Команда из n роботов сражается с n призраками. Каждый робот вооружён протонным ружьём, луч которого уничтожает призрак. Луч движется по прямой и обрывается, встретив призрак. Тактика роботов такова: каждый выбирает себе призрак (при этом образуется n пар робот — призрак), а затем все роботы одновременно выпускают лучи каждый в свое приведение.

При этом нельзя, чтобы выпускаемые роботами лучи пересеклись.

Считаем, что положение каждого робота и каждого призрака — заданная точка плоскости, и никакие три из этих точек не лежат на одной прямой.

a. Доказать, что всегда существует прямая, соединяющая одного из роботов с одним из призраков, для которой число роботов по одну сторону от неё совпадает с числом призраков с той же стороны. Придумайте, как найти такую прямую за время $O(n \lg n)$.

b. Постройте алгоритм, за время $O(n^2 \lg n)$ *группирующий роботов и приведений в пары так, чтобы выпускаемые роботами лучи не пересекались.*

35-4. Распределение с малой оболочкой.

Пусть мы хотим построить выпуклую оболочку множества случайных точек плоскости, подчиняющихся некоторому распределению вероятностей (различные точки независимы). Для некоторых распределений математическое ожидание числа вершин выпуклой оболочки есть $O(n^{1-\varepsilon})$ *для некоторой константы* $\varepsilon > 0$ *(здесь n — число точек). Такие распределения мы будем называть* распределениями с малой оболочкой *(sparse-hulled distributions). Вот примеры таких распределений:*

Точки равномерно распределены внутри круга единичного радиуса. Выпуклая оболочка имеет в среднем $\Theta(n^{1/3})$ *вершин.*

Точки равномерно распределены внутри фиксированного выпуклого многоугольника. Выпуклая оболочка содержит в среднем $\Theta(\lg n)$ *вершин. (Константа зависит от числа вершин многоугольника.)*

Точки подчинены двумерному нормальному распределению. Выпуклая оболочка содержит в среднем $\Theta(\sqrt{\lg n})$ *вершин.*

a. Даны два выпуклых многоугольника (возможно, пересекающихся) с n_1 *и* n_2 *вершин. Придумайте способ построить выпуклую оболочку всех* n_1+n_2 *вершин за время* $O(n_1+n_2)$. *(Многоугольники могут пересекаться.)*

b. Постройте алгоритм, который отыскивает выпуклую оболочку n независимых точек, подчинённых некоторому распределению с малой оболочкой. Математическое ожидание времени работы т алгоритма должно быть $O(n)$. *(Указание. Рекурсивно ищем выпуклую оболочку для двух половин множества, а затем соединяем эти выпуклые оболочки в одну.)*

Замечания

В этой главе мы лишь слегка коснулись задач вычислительной геометрии. Книги по вычислительной геометрии написали Препарата и Шамос [160], а также Эдельсбруннер [60].

Хотя сама геометрия имеет тысячелетнюю историю, алгоритмические задачи в геометрии стали рассматривать лишь недавно. Как отмечают Препарата и Шамос, впервые сложностью *геометрических задач заинтересовался Лемуан (E. Lemoine) в 1902 году. Изучая построения при помощи циркуля и линейки, он выделил пять элементарных действий (установить одну из ножек циркуля в данную точку, установить одну из ножек циркуля на данной прямой, нарисовать круг, приложить край линейки так, чтобы он проходил через данную точку, провести прямую). Лемуан интересовался минимальным числом действий, требуемых для решения данной задачи на построение.*

Алгоритм раздела 35.2, выясняющий, есть ли в множестве отрезков пересекающиеся, разработали Шамос и Хой [176].

Грэхем (Grahm) описывает "просмотр Грэхема" в [91]. Алгоритм с "заворачиванием" принадлежит Джарвису [112]. Используя так называемые разрешающие деревья в качестве вычислительной модели, Яо доказал [205] нижнюю оценку $\Omega(n\ln n)$ *на время работы любого алгоритма, находящего выпуклую оболочку.*

Если интересоваться временем работы как функцией от числа всех точек (n) *и от числа точек выпуклой оболочки* (h)*, наиболее быстрым является метод "стрижки и поиска" (prune-and-search method), который придумали Киркпатрик и Зайдель [120]; требующий времени* $O(n\lg h)$*. Этот алгоритм является асимптотически оптимальным.*

Описанный нами алгоритм поиска пары ближайших точек (время работы $O(n\ln n)$*) предложил Шамос (см. Препарата и Шамос [160]). Препарата и Шамос показали, что этот алгоритм асимптотически оптимален в модели разрешающих деревьев.*

36

NP-полнота

Все рассмотренные нами ранее алгоритмы были полиномиальными (работали за полиномиальное время). Это значит, что время работы алгоритма на входе длины n составляло не более $O(n^k)$ для некоторой константы k (не зависящей от длины входа). Естественно поинтересоваться, всякая ли задача может быть решена за полиномиальное время.

Ответ на этот вопрос сугубо отрицательный — некоторые задачи вообще не могут быть решены никаким алгоритмом. Классический пример такой задачи — "проблема остановки" (выяснить, останавливается ли данная программа на данном входе). Кроме того, существуют задачи, для которых существует решающий их алгоритм, но любой такой алгоритм работает долго — время его работы не есть $O(n^k)$ ни для какого фиксированного числа k.

Если мы хотим провести пусть грубую, но формальную границу между "практическими" алгоритмами и алгоритмами, представляющими лишь теоретический интерес, то класс алгоритмов, работающих за полиномиальное время, является разумным первым приближением.

В этой главе мы рассмотрим класс задач, называемых " NP-полными". Для этих задач не найдены полиномиальные алгоритмы, однако не доказано, что таких алгоритмов не существует. Изучение NP-полных задач связано с так называемым вопросом $\mathrm{P} \neq \mathrm{NP}$*. Этот вопрос был поставлен в 1971 году и является сейчас одной из наиболее интересных и сложных проблем теории вычислений.*

Большинство специалистов полагают, что NP-полные задачи нельзя решить за полиномиальное время. Дело в том, что если хотя бы для одной NP-полной задачи существует решающий ее полиномиальный алгоритм, то и для всех NP-полных задач такие алгоритмы существуют. В настоящее время известно очень много NP-полных задач. Все попытки найти для них полиномиальные алгоритмы оказались безуспешными. По-видимому, таких алгоритмов нет вовсе.

Зачем программисту знать о NP-полных задачах? Если для некоторой задачи удается доказать её NP-полноту, есть основания считать её практически неразрешимой. В этом случае лучше потратить время на построение приближенного алгоритма (гл. 37), чем продолжать искать быстрый алгоритм, решающий её точно.

Многие интересные и практически важные задачи являются NP-полными, хотя на вид ничуть не сложнее, чем задача сортировки, задача о кратчайшем пути в графе или о максимальном потоке в сети (для этих задач полиномиальные алгоритмы существуют).

В разделе 36.1 мы формализуем понятие задачи и определим класс P, состоящий из "задач" (или, как говорят, "языков"), разрешимых за полиномиальное время. В разделе 36.2 будет определен класс NP, состоящий из задач, решение которых может быть проверено за полиномиальное время. Вопрос о совпадении этих классов и есть та центральная проблема теории сложности вычислений, о которой мы говорили.

В разделе 36.3 мы введем понятие сводимости за полиномиальное время, дадим определение NP-полной задачи и установим NP-полноту задачи о выполнимости. После этого в разделе 36.4 мы докажем NP-полноту некоторых других задач, сведя к ним задачу о выполнимости. В разделе 36.5 будет доказана NP-полнота ещё нескольких задач.

36.1 Полиномиальное время

Как мы уже говорили, понятие полиномиально разрешимой задачи принято считать уточнением идеи "практически разрешимой" задачи. Чем объясняется такое соглашение?

Во-первых, используемые на практике полиномиальные алгоритмы обычно действительно работают довольно быстро. Конечно, трудно назвать практически разрешимой задачу, котрая требует времени n^{100}*. Однако полиномы такой степени в реальных задачах почти не встречаются.*

Второй аргумент в пользу рассмотрения класса полиномиальных алгоритмов — тот факт, что объём этого класса не зависит от выбора конкретной модели вычислений (для достаточно широкого класса моделей). Например, класс задач, которые могут быть решены за полиномиальное время на последовательной машине с произвольным доступом (RAM), совпадает с классом задач, полиномиально разрешимых на машинах Тьюринга (определение которых можно найти в книге Хопкрофта и Ульмана [104] или Льюиса и Пападимитриу [139]). Класс будет тем же и для моделей параллельных вычислений, если, конечно, число процессоров полиномиально зависит от длины входа.

В-третьих, класс полиномиально разрешимых задач обладает естественными свойствами замкнутости. Например, композиция двух полиномиальных алгоритмов (выход первого алгоритма подается на вход второго) также работает полиномиальное время. Объясняется это тем, что сумма, произведение и композиция многочленов снова есть многочлен.

Абстрактные задачи

Мы принимаем следующую абстрактную модель вычислительной задачи. Будем называть абстрактной задачей (abstract problem) произвольное бинарное отношение Q между элементами двух множеств — множества условий (instances) I и множества решений (solutions) S. Например в задаче SHORTEST-PATH (поиск кратчайшего пути между двумя заданными вершинами некоторого неориентированного графа $G = (V, E)$) условием (элементом I) является тройка, состоящая из графа и двух вершин, а решением (элементов S) — последовательность вершин, составляющих требуемый путь в графе. При этом один элемент множества I может находиться в отношении Q с несколькими элементами множества S (если кратчайших путей между данными вершинами несколько)

В теории NP-полноты рассматриваются только задачи разрешения (decision problems) — задачи, в которых требуется дать ответ ”да” или ”нет” на некоторый вопрос. Формально задачу разрешения можно рассматривать как функцию, отображающую множество условий I в множество $\{0, 1\}$ ($1 =$ ”да”, $0 =$ ”нет”). Многие задачи можно там или иным способом преобразовать к такому виду. Например, с задачей поиска кратчайшего пути в графе связана задача разрешения PATH: ”по заданному графу $G = (V, E)$, паре вершин $u, v \in V$ и натуральному числу k определить, существует ли в G путь между вершинами u и v, длина которого не превосходит k”. Пусть четверка $i = \langle G, u, v, k \rangle$ является условием задачи PATH. Тогда $\text{PATH}(i) = 1$, если длина кратчайшего пути между вершинами u и v не превосходит k, и $\text{PATH}(i) = 0$ в противном случае.

Часто встречаются задачи оптимизации (optimization problems), в которых требуется минимизировать или максимизировать значение некоторой величины. Прежде чем ставить вопрос о NP-полноте таких задач, их следует преобразовать в задачу разрешения. Обычно в качестве такой задачи разрешения рассматривают задачу проверки, является ли некоторое число верхней (или нижней) границей для оптимизируемой величины. Так, например, мы перешли от задачи оптимизации SHORTEST-PATH к задаче разрешения PATH, добавив в условие задачи границу длины пути k.

Если после этого получается NP-полная задача, то тем самым установлена трудность исходной задачи. В самом деле, если для оптимизационной задачи имеется быстрый алгоритм, то и полученную из неё задачу разрешения можно решить быстро (надо просто сравнить ответ этого алгоритма с заданной границей). Таким образом, теорию NP-полноты можно использовать и для исследования задач оптимизации.

Представление данных

Прежде чем подавать на вход алгоритма исходные данные (то есть элемент множества I), надо договориться о том, как они представляются в "понятном для компьютера виде"; мы будем считать, что исходные данные закодированы последовательностью битов. Формально говоря, представлением *элементов некоторого множества S называется отображение e из S во множество битовых строк. (Разумеется, вместо битовых строк можно было бы рассматривать последовательности символов любого другого конечного алфавита, имеющего не менее двух символов.) Например, натуральные числа* $0, 1, 2, 3, \ldots$ *обычно представляют битовыми строками* $0, 1, 10, 11, 100, \ldots$ *(при этом, например,* $e(17) = 10001$*). В компьютерах для представления букв и других символов используют код ASCII (реже — EBCDIC). Так,* $e(A) = 1000001$ *в коде ASCII.*

Фиксировав представление данных, мы превращаем абстрактную задачу в строковую, *для который входным данным является битовая строка, представляющая исходное данное абстрактной задачи. Будем говорить, что алгоритм* решает *(solves) строковую задачу за время* $O(T(n))$, *если на входном данном (битовой строке) i длины n алгоритм работает время* $O(T(n))$. *Строковая задача называется* полиномиальной *(polynomial-time solvable), существует константа k и алгоритм, решающий эту задачу за время* $O(n^k)$.

Сложностным классом *P называется множество всех строковых задач разрешения, которые могут быть решены за полиномиальное время.*

Желая определить понятие полиномиальной абстрактной задачи, мы сталкиваемся с тем, что возможны различные представления данных. Для каждого представления e множества I входов абстрактной задачи Q мы получаем свою строковую задачу, которую мы в дальнейшем обозначаем $e(Q)$. *Вообще говоря, при одном представлении может получиться полиномиальная строковая задача, а при другом — нет. (Тут есть ещё одна — чисто техническая — деталь: некоторые битовые строки могут не представлять никаких исходных данных; что требуется в этом случае от алгоритма, решающего строковую задачу? Будем считать, что он должен давать в этом случае ответ* 0, *то есть что свойство* $e(Q)$ *ложно для таких входов.)*

Вернёмся к вопросу о том, как зависит полиномиальность задачи от выбранного представления. Пусть, например, входом задачи является натуральное число k, и время работы алгоритма есть $\Theta(k)$. Если число k представить в системе счисления с основанием 1 (unary representation), то есть в виде последовательности k единиц, то время работы алгоритма на входе длины n будет равно $O(n)$, и алгоритм будет полиномиальным. При более естественном двоичном представлении числа k время работы на входе длины n (то есть на n-битовом числе) будет равно $\Theta(2^n)$, и алгоритм не будет полиномиальным.

Однако на практике (если исключить такие "дорогие" способы представления, как система счисления с основанием 1) естественные способы представления оказываются обычно эквивалентными, поскольку они могут быть быстро (полиномиально) преобразованы друг в друга. Например, двоичную запись числа можно преобразовать в троичную и обратно за полиномиальное время.

Будем говорить, что функция $f : \{0,1\}^* \to \{0,1\}^*$ вычислима за полиномиальное время *(is polynomial-time computable), если существует полиномиальный алгоритм A, который для любого $x \in \{0,1)^*$ выдает результат $f(x)$.*

Рассмотрим теперь множество I условий произвольной абстрактной задачи разрешения. Два представления e_1 и e_2 этого множества называются полиномиально связанными *(polynomially related), если существуют две вычислимые за полиномиальное время функции ода f_{12} и f_{21}, для которых $f_{12}(e_1(i)) = e_2(i)$ и $f_{21}(e_2(i)) = e_1(i)$ для всякого $i \in I$. Это значит, что e_1-представление входа может быть за полиномиальное время получено из e_2-представления и наоборот. В этом случае не имеет значения, какое из двух полиномиально связанных представлений выбрать, как показывает следующая лемма.*

Лемма 36.1

Пусть Q — абстрактная задача разрешения с множеством условий I, а e_1 и e_2 — полиномиально связанные представления для элементов множества I. Предположим, что множество всех строк, которые являются e_1-представлениями элементов Q, разрешимо за полиномиальное время, и что аналогичное свойство выполнено для представления e_2. Тогда свойства $e_1(Q) \in \mathrm{P}$ и $e_2(Q) \in \mathrm{P}$ равносильны.

Доказательство. Утверждение симметрично, так что достаточно доказать его в одну сторону. Предположим, что задача $e_1(Q)$ разрешима за время $O(n^k)$ для некоторого фиксированного числа k. По предположению для всякого условия $i \in I$ представление $e_1(i)$ может быть получено из представления $e_2(i)$ за время $O(n^c)$ (где c — некоторая константа, $n = |e_2(i)|$). Для решения задачи $e_2(Q)$, получив на вход $e_2(i)$, мы сперва вычислим

$e_1(i)$, а затем применим алгоритм, разрешающий $e_1(Q)$, к строке $e_1(i)$. Сколько времени займёт наше вычисление? Преобразование $e_2(i)$ в $e_1(i)$ требует полиномиального времени. Следовательно, $|e_1(i)| = O(n^c)$, поскольку длина выхода алгоритма не превосходит времени его работы. Решение задачи с условием $e_1(i)$ занимает $O(|e_1(i)|^k) = O(n^{ck})$ времени. Итак, время вычисления оказалось полиномиальным. (Мы пропустили важный момент: получив на входе некоторую строку, мы должны сначала проверить, что она является e_2-представлением некоторого входа; по предположению это можно сделать за полиномиальное время.)

Мы не будем подробно описывать используемое представление в конкретных задачах, считая, что оно выбрано достаточно разумно и экономно (целые числа задаются двоичной записью, конечные множества — списком элементов и т.п.) Представление объекта будем обозначать угловыми скобками: $\langle G \rangle$ — это стандартное представление объекта G. При этом множество всех строк, являющихся представлениями, оказывается полиномиальным, а различные "разумные" способы представления данных оказываются полиномиально связанными, так что можно воспользоваться леммой 36.1 и не описывать представление детально, если нас интересует лишь вопрос о полиномиальности задачи. Таким образом, в дальнейшем мы не будем делать различия между абстрактной задачей и ее строковым представлением, как это обычно и делают (кроме тех редких задач, в которых стандартное представление не очевидно — для них выбор представления может сильно повлиять на сложность решения задачи).

Формальные языки

Для задач разрешения удобно использовать терминологию теории формальных языков. Алфавитом *(alphabet) Σ называется любой конечный набор символов.* Языком *L над алфавитом Σ (language L over Σ) называется произвольное множество строк символов из алфавита Σ (такие строки называют* словами *в алфавите Σ) Например, можно рассмотреть $\Sigma = \{0, 1\}$ и язык $L = \{10, 11, 101, 111, 1011, 1101, 10001, \ldots\}$, состоящий из двоичных записей простых чисел. Мы будем обозначать символом ε* пустое слово *(empty string), не содержащее символов, а символом $\emptyset$ —* пустой язык *(empty language), не содержащий слов. Язык, состоящий из всех строк в алфавите Σ, обозначается Σ^*. Например, при $\Sigma = \{0, 1\}$ имеем $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$. Таким образом, всякий язык L над Σ является подмножеством множества Σ^*.*

Имеется несколько стандартных операций над языками. Операции объединения *(union) и* пересечения *(intersection) языков определяются как обычные операции объединения и пересечения множеств.* Дополнением *(complement) языка L называют язык $\bar{L} = \Sigma^* \setminus L$.* Конкатенацией *(concatenation), или* соединением, *двух язы-*

ков L_1 и L_2 называется язык

$$L = \{x_1x_2 : x_1 \in L_1, x_2 \in L_2\}.$$

Замыканием *(closure) языка L называется язык*

$$L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \dots ,$$

*где L^k — язык, полученный k-кратной конкатенацией языка L с самим собой. Операция замыкания называется также $\star$-*операцией Клини *(Kleene star).*

Теперь можно сказать, что задача разрешения (точнее, соответствующая ей строковая задача разрешения) является языком над алфавитом $\Sigma = \{0,1\}$. Например, задаче PATH соответствует язык PATH $= \{\langle G,u,v,k\rangle :$ $G = (V,E)$ — неориентированный граф, , $u,v \in V$; $k \geqslant 0$ — целое число, и в графе G существует путь из u в v, длина которого не превосходит $k\}$.

(Мы будем использовать одно и то же название — в данном случае PATH — для обозначения задачи и соответствующего языка.)

Продолжим знакомство с терминологией теории формальных языков. Говорят, что алгоритм A допускает *(accepts) строку $x \in \{0,1\}^*$, если на входе x алгоритм выдает результат 1 ($A(x) = 1$). Алгоритм A* отвергает *(rejects) слово x, если $A(x) = 0$. (Заметим, что алгоритм может не остановиться на входе x или дать ответ, отличный от 0 и 1. В этом случае он и не допускает, и не отвергает слово x.) Алгоритм A допускает (accepts) язык L, если алгоритм допускает те и только те слова, которые принадлежат языку L.*

Алгоритм A, допускающий некоторый язык L, не обязан отвергать всякое слово $x \notin L$. Если алгоритм допускает все слова из L, а все остальные слова отвергает, говорят, что что A распознаёт *(decides) язык L. Язык L* допускается за полиномиальное время *(is accepted in polynomial time), если имеется алгоритм A, который допускает данный язык, причем всякое слово $x \in L$ допускается алгоритмом за время $O(n^k)$, где n — длина слова x, а k — некоторое не зависящее от x число. Язык L называется* распознаётся за полиномиальное время *(is decided in polynomial time), если некоторый алгоритм A распознаёт данный язык, причем время работы алгоритма на каждом слове длины n не больше $O(n^k)$.*

Рассмотренный нами язык PATH допускается за полиномиальное время. Нетрудно построить алгоритм, который методом поиска в ширину за полиномиальное время находит кратчайший путь между вершинами u и v в графе G, а затем сравнивает длину найденного пути с данным в условии числом k. Если длина пути не превосходит k, алгоритм выдаёт 1 и останавливается.

В противном случае алгоритм зацикливается, не выдавая никакого ответа. Ясно, что такой алгоритм допускает, но не распознаёт язык PATH. Однако легко исправить описанный алгоритм таким образом, чтобы слова, не принадлежащие языку, отвергались (если длина кратчайшего пути превосходит k, надо отвергать входное слово). Такой алгоритм допускает и распознаёт язык PATH.

Отметим, что для некоторых языков (например, для множества всех программ, заканчивающих свою работу) есть допускающий, но нет распознающего алгоритма.

Мы уже дали определение класса задач P. Ниже мы определим также класс NP. В теории сложности вычислений рассматриваются многие другие сложностные классы *(complexity classes). Например, имеется важный класс PSPACE, состоящий из задач, решаемых алгоритмами с использованием памяти полиномиального размера, или класс EXP, состоящий из задач, которые можно решить за время* $O(2^{n^k})$. *Неформально сложностной класс можно определить как семейство языков, для которых распознающие алгоритмы имеют заданную* меру сложности *(например заданное время работы). Мы не даём точного определения, отсылая заинтересованного читателя работе Хартманиса и Стирнса [95].*

Теперь можно переформулировать определение класса P так:
$\mathrm{P} = \{L \subset \{0,1\}^* :$ *существует алгоритм* A, *распознающий язык* L*за полиномиальное*

На самом деле в данной ситуации нет разницы между языками, допускаемыми и распознаваемыми за полиномиальное время.

Теорема 36.2

$$\mathrm{P} = \{L : L \text{ допускается за полиномиальное время}\}$$

Доказательство

Если язык распознаётся некоторым алгоритмом, то он и допускается тем же алгоритмом. Остаётся доказать, что если язык L допускается полиномиальным алгоритмом A, то он распознаётся некоторым (возможно, другим) полиномиальным алгоритмом A'. *Пусть алгоритм A допускает язык L за время* $O(n^k)$. *Это значит, что существует константа c, для которой A допускает любое слово длины n из L, сделав не более* $T = cn^k$ *шагов. (Формально говоря, это верно для достаточно длинных слов x; мы опускаем очевидные детали.)*

Новый алгоритм A' *моделирует работу алгоритма A и считает число шагов этого алгоритма, сравнивая его с известной границей T. Если за время T алгоритм A допускает слово x, алгоритм* A' *также допускает это слово и выдаёт* 1. *Если же A не допускает x за указанное время, то алгоритм* A' *прекращает моделирование и отвергает слово (выдаёт* 0*). Замедление работы за счёт моделирования и подсчёта шагов не так уж и велико и* *оставляет время работы полиномиальным.*

Заметим, что доказательство теоремы 36.2 неконструктивно: если мы имеем полиномиальный алгоритм A, допускающий язык язык L, но не знаем, каким именно полиномом ограничено время его работы, мы не можем построить алгоритма A', а можем лишь утверждать, что такой алгоритм существует.

Упражнения

36.1-1

Определим задачу оптимизации LONGEST-PATH-LENGTH. Условиями являются тройки, состоящие из неориентированного графа и двух его вершин; её решение — это самый длинный простой путь в графе, концами котрого являются данные вершины. Рассмотрим соответствующую задачу разрешения LONGEST-PATH $= \{\langle G,u,v,k\rangle$: $G = (V,E)$ — *неориентированный граф,* $u, v \in V$, $k \geqslant 0$ — *целое число, и существует простой путьиз* u *в* v, *длины не менее* $k\}$. *Покажите, что задача оптимизации LONGEST-PATH-LENGTH может быть решена за полиномиальное время в том и только том случае, если задача разрешения LONGEST-PATH принадлежит классу P.*

36.1-2

Определите формально задачу нахождения наибольшего простого цикла в неориентированном графе. Сформулируйте соответствующую строковую задачу разрешения (язык).

36.1-3

Рассмотрите два представления для графа — с помощью матрицы инцидентности (закодированной последовательностью битов) и с помощью списков рёбер (также надлежащим образом закодированных). Объясните, почему два данных представления полиномиально связаны.

36.1-4

Является ли алгоритм динамического программирования для 0-1-*задачи о рюкзаке из упражнения 17.2-2 полиномиальным? Объясните свой ответ.*

36.1-5

Предположим, что некоторый язык L допускается за полиномиальное время некоторым алгоритмов. Может ли этот алгоритм работать более чем полиномиальное время на словах, не принадлежащих L? (Сравните свой ответ с утверждением теоремы 36.2.)

36.1-6

Покажите, что алгоритм, который содержит фиксированное (не зависящее от входа) число вызовов процедуры, работающей за полиномиальное время, сам работает полиномиальное время. Если же алгоритм делает полиномиальное число вызовов такой процедуры, то общее время работы может быть экспоненциаль-

ным. Почему?

36.1-7

Покажите, что класс P как множество языков замкнут относительно объединения, пересечения, дополнения, конкатенации и $\star$-операции Клини. Это значит, что из $L_1, L_2 \in \mathrm{P}$ следует $L_1 \cup L_2 \in \mathrm{P}$ и т.д.

36.2 Проверка принадлежности языку и класс NP

В предыдущем разделе мы рассматривали задачу разрешения PATH (выяснить, есть ли в данном графе G между данными двумя вершинами u и v путь длины не более k). Эта задача оказалась полиномиальной (и решается с помощью алгоритма поиска в ширину). Представим себе, однако, что мы ничего не знаем про поиск в ширину. Тогда задача PATH будет для нас трудной: видя граф G, вершины u и v и знаю число k, мы не будем знать, есть ли искомый путь, пока нам кто-нибудь его не покажет. Но если нам его покажут, то мы можем без труда проверить, что путь этот — искомый.

Именно такая ситуация имеет место для задач из класса NP, о которых мы будем говорить в этом разделе.

Гамильтонов цикл

Задача поиска гамильтонова цикла в графе изучается уже более ста лет. Бонди и Мёрти [131] цитируют письмо Гамильтона (W.R.Hamilton), в котором описана такая игра: первый игрок отмечает в додекаэдре (рис. 26.1(a)) путь из пяти идущих друг за другом вершин, а второй игрок должен дополнить этот путь до простого цикла, проходящего через все вершины додекаэдра.

Вообще гамильтоновым циклом *(hamiltonian cycle) в неориентированном графе $G = (V, E)$ называется простой цикл, который проходит через все вершины графа. Графы, в которых есть гамильтонов цикл, называют* гамильтоновыми *(hamiltonian graph)*

Додекаэдр (проекция которого изображена на рис. 35.1(a)) — гамильтонов граф; серым цветом показан один из гамильтоновых циклов. Не все графы гамильтоновы: на рис. 36.1(b) показан двудольный граф с нечётным числом вершин; легко видеть (упр. 36.2-2), что такой граф не может иметь гамильтонова цикла.

Задача о гамильтоновом цикле *(hamiltonian path problem) состоит выяснении, имеет ли данный граф G гамильтонов цикл.*

36.1 (a) Додекаэдр (вид из точки, расположенной недалеко от центра одной из граней); серые рёбра образуют гамильтонов цикл.
(b) Двудольный граф с нечётным числом вершин — такой граф не гамильтонов.

Формально говоря,

$$\textit{HAM-CYCLE} = \{\langle G\rangle : G \text{ — гамильтонов граф}\}$$

Как решать такую задачу? Можно перебрать все перестановки вершин данного графа и проверить, является ли хотя бы одна из них гамильтоновым циклом. Оценим время работы такого алгоритма. Если мы используем представление графа с помощью матрицы инцидентности, то число вершин m в графе будет $\Omega(\sqrt{n})$, где $n = |\langle G\rangle|$ — длина представления графа G. Имеется $m!$ различных перестановок вершин графа, и время работы алгоритма равно $\Omega(m!) = \Omega(\sqrt{n}!) = \Omega(2^{\sqrt{n}})$, то есть не является полиномиальным. Таким образом, наивный алгоритм не даёт эффективного решения задачи. На самом деле, как мы покажем в разделе 36.1, задача о гамильтоновом цикле является NP-полной, и потому можно предполагать, что полиномиального алгоритма для неё вообще не существует.

Проверка принадлежности языку

Пусть вы заключили пари с приятелем, который утверждает, что (нарисованный перед вами на доске) граф является гамильтоновым. При этом вы не можете быстро проверить, так это или нет. Тем не менее приятель может выиграть пари, если каким-то образом отгадает гамильтонов цикл и предъявит его вам: проверка того, что данный цикл является гамильтоновым, проста. Нужно лишь проверить, что предъявленный цикл проходит через все вершины графа, и что он действительно идёт по рёбрам. Итак, доказательство существования гамильтонова цикла в графе (состоящее в предъявлении гамильтонова пути) можно проверить за полиномиальное время.

Назовем проверяющим алгоритмом *(verification algorithm) алгоритм A с двумя аргументами; первый аргумент мы будем называть (как и раньше) входной строкой, а второй —* сертификатом *(certificatie). Мы говорим, что алгоритм A с двумя аргументами* допускает *вход x (A verifies an input string x), если существует сертификат y, для которого $A(x, y) = 1$.* Языком, проверяемым алгоритмом *A (language verified by A), мы назовём язык*

$$L = \{x \in \{0,1\}^* : \text{ существует } y \in \{0,1\}^*, \text{ для которого} A(x, y) = 1\}.$$

Другими словами, алгоритм A проверяет язык L, если для любой строки $x \in L$ найдется сертификат y, с помощью которого A может проверить принадлежность x к языку L, а для $x \notin L$ такого сертификата нет. Например, в задаче HAM-CYCLE сертификатом была последовательность вершин, образующая гамильтонов цикл.

Сложностной класс NP

Сложностной класс NP *(complexity class NP) — это класс языков, для которых существуют проверяющие алгоритмы, работающие полиномиальное время, причём длина сертификата также ограничена некоторым полиномом. Более точно, язык L принадлежит классу* NP, *если существует такой полиномиальный алгоритм A с двумя аргументами и такая многочлен $p(x)$ с целыми коэффициентами, что*

$$L = \{x \in \{0,1\}^* : \textit{ существует сертификат } y, \textit{ для которого } |y| \leqslant p(|x|) \textit{ и } A(x,y) = 1$$

В этом случае мы говорим, что алгоритм A проверяет язык L за полиномиальное время *(A verifies language L in polynomial time). Заметим, что, формально говоря, из этого не следует, что A проверяет язык L в смысле ранее данного определения, так как для $x \notin L$ могут существовать длинные сертификаты, которые отбрасываются при новом определении, но препятствуют старому.*

Несколько слов о названии: сокращение NP происходит от английских слов Nondeterministic Polynomial (time), что переводится как недетерминированное полиномиальное время. Первоначально класс NP определялся в терминах так называемых недетерминированных вычислений (см. книгу Хопкрофта и Ульмана [104]).

Мы уже знаем одну задачу из класса NP — это задача HAM-CYCLE. Кроме того, всякая задача из P принадлежит также и NP. Действительно, если есть полиномиальный алгоритм, распознающий язык, то легко построить проверяющий алгоритм для того же языка — проверяющий алгоритм может просто игнорировать свой второй аргумент (сертификат). Таким образом $\mathrm{P} \subseteq \mathrm{NP}$.

В данное время неизвестно, совпадают ли классы P и NP, но большинство специалистов полагает, что нет. Интуитивно класс P можно представлять себе как класс задач, которые можно быстро решить, а класс NP — как класс задач, решение которых может быть быстро проверено. На практике решить самому задачу часто намного труднее, чем проверить уже готовое решение, особенно если время работы ограничено. По аналогии можно думать, что в классе NP имеются задачи, которые нельзя решить за полиномиальное время.

Есть и более серьёзные причины полагать, что $\mathrm{P} \neq \mathrm{NP}$ *— прежде всего это существование "NP-полных" задач (раздел 36.3).*

Кроме проблемы $\mathrm{P} = \mathrm{NP}$, *остаются открытыми и многие другие вопросы о классе* NP. *Так, несмотря на огромные усилия исследователей, не известно, замкнут ли класс* NP *относительно дополнения, то есть верно ли, что из* $L \in \mathrm{NP}$ *следует* $\bar{L} \in \mathrm{NP}$. *Мы можем определить сложностной класс* $\mathrm{co} - \mathrm{NP}$ *как множество языков L, для котрых* $\bar{L} \in \mathrm{NP}$. *Вопрос о замкнутости класса* NP

36.2 Четыре возможных ситуации (имеется в виду, что все показанные на рисунке области непусты).
(a) Классы P, NP и $co-NP$ совпадают. (Большинство экспертов считает это наименее вероятным.)
(b) Класс NP замкнут относительно дополнения (NP=co-NP), но не совпадает с P.
(c) Класс NP не замкнут относительно дополнения и в пересечении с co-NP даёт P.
(d) Класс NP не замкнут относительно дополнения; пересечение $\mathrm{NP} \cap \mathrm{co} - \mathrm{NP}$ больше, чем P.

относительно дополнения можно переформулировать так: равны ли классы NP и co-NP?

Поскольку класс P *замкнут относительно дополнения (упр. 36.1-7),* $\mathrm{P} \subseteq \mathrm{NP} \cap \mathrm{co} - \mathrm{NP}$. *В то же время остаётся неизвестным, верно ли равенство* $\mathrm{P} = \mathrm{NP} \cap \mathrm{co} - \mathrm{NP}$. *На рис. 36.2 показаны четыре возможных ситуации.*

Увы, наши знания о соотношении классов P и NP далеко не полны (говоря прямо, мы вообще ничего не знаем). Но уже понятие NP-полноты является важным средством классификации задач; как мы увидим, оно сводит вопрос о сложности данной задачи к общему (пусть и не решённому) вопросу о соотношении классов P и NP.

Упражнения

36.2-1

Рассмотрим язык GRAPH-ISOMORPHISM=$\{\langle G_1, G_2\rangle :$ *графы* G_1 *и* G_2 *изоморфны*$\}$. *Докажите, что данный язык принадлежит классу NP (постройте полиномиальный алгоритм, проверяющий этот язык).*

36.2-2

Докажите, что неориентированный двудольный граф с нечётным числом вершин не гамильтонов.

36.2-3

Покажите, что если HAM-CYCLE лежит в P, то за полиномиальное время можно не только проверить, существует ли гамильтонов цикл, но и найти его.

36.2-4

Докажите, что класс языков NP замкнут относительно объединения, пересечения, конкатенации и $\star$*-операции Клини.*

36.2-5

Докажите, что любой язык из NP распознается некоторым алгоритмом за время $2^{O(n^k)})$, *где* n *— длина входа, а* k *— не зависящая от* n *константа.*

36.2-6

Гамильтоновым путём *в графе называется путь, котрый прохо-*

дит через каждую вершину графа ровно один раз. Докажите, что язык HAM-PATH=$\{\langle G,u,v\rangle:$ *в графе G существует гамильтонов путь из вершины принадлежит NP.*

36.2-7

Покажите, что для ориентированных графов без циклов можно за полиномиальное время выяснить, существует ли гамильтонов путь.

36.2-8

Пусть φ *— булева формула, построенная из булевых переменных* $x_1, x_2, \ldots,$

x_n *с помощью операций отрицания* ($\neg$)*, конъюнкций (И,* $\wedge$*), дизъюнкций (ИЛИ,* $\vee$*) и скобок. Формула* φ *называется* тавтологией *(tautology), если она истинна для всех значений переменных. Определим язык TAUTOLOGY, состоящий из всех тавтологий. Покажите, что TAUTOLOGY принадлежит co-NP.*

36.2-9

Докажите, что $\mathrm{P} \subseteq \mathrm{co}-\mathrm{NP}$

36.2-10

Докажите, что если $\mathrm{NP} \neq \mathrm{co}-\mathrm{NP}$*, то* $\mathrm{P} \neq \mathrm{NP}$.

36.2-11

Пусть G — связный неориентированный граф, имеющий не меньше трёх вершин. Рассмотрим граф G^3*, который получится, если соединить рёбром каждую пару вершин графа G, которые соединены в G путём длины не больше 3. Докажите, что граф* G^3 *гамильтонов. (Указание: постройте покрывающее дерево графа G и примените математическую индукцию.)*

36.3 NP-полнота и сводимость

По-видимому, наиболее убедительным аргументом в пользу того, что классы P и NP различны, является существование так называемых NP-полных задач. Этот класс обладает удивительным свойством: если какая-нибудь NP-полная задача разрешима за полиномиальное время, то и все задачи из класса NP разрешимы за полиномиальное время, то есть P=NP. Несмотря на многолетние исследования, ни для одной NP-полной задачи не найден полиномиальный разрешающий алгоритм.

В частности, задача HAM-PATH (о гамильтоновом цикле) является NP-полной. Таким образом, если научиться решать её за полиномиальное время, то и для всех задач класса NP существовали бы полиномиальные алгоритмы. Переформулировка: $\mathrm{NP} \setminus \mathrm{P}$ *непусто, то HAM-PATH*$\in \mathrm{NP} \setminus \mathrm{P}$.

Неформально говоря, NP-полные языки являются самыми " трудными" в классе NP. При этом трудность языков можно *сравнивать, сводя один язык к другому. В этом разделе мы даём определение сводимости, затем определяем класс NP-полных языков и устанавливаем полноту одного конкретного языка, CIRCUIT-SAT. В разделе 36.5 мы используем метод сведения для доказательства NP-полноты многих других задач.*

Сводимость

Говоря неформально, задача Q сводится к задаче Q', если задачу Q можно решить для любого входа, считая известным решение задачи Q' для какого-то другого входа. Например, задача решения линейного уравнения сводится к задаче решения квадратного уравнения (линейное уравнение можно превратить в квадратное, добавив фиктивный старший член). Если задача Q сводится к задаче Q', то любой алгоритм, решающий Q', можно использовать для решения задачи Q, то есть Q "не труднее" Q'.

Дадим формальное определение. Говорят, что язык L_1 сводится за полиномиальное время *(is polynomial-time reducible) к языку L_2 (запись: $L_1 \leqslant_{\mathrm{P}} L_2$), если существует такая функция $f : \{0,1\}^* \to \{0,1\}^*$, вычислимая за полиномиальное время, что для любого $x \in \{0,1\}^*$,*

$$x \in L_1 \Leftrightarrow f(x) \in L_2 \tag{36.1}$$

Функцию f называют сводящей функцией (reduction function). а полиномиальный алгоритм F, вычисляющий функцию f — сводящим алгоритмом (reduction algorithm).

Рисунок 36.3 иллюстрирует определение сводимости языка L_1 к языку L_2 за полиномиальное время. Каждый язык есть подмножество множества $\{0,1\}^*$. Сводящая функция переводит любое слово $x \in L_1$ в слова $f(x) \in L_2$, а слово $x \notin L_1$ — в слово $f(x) \notin L_2$. Поэтому, если мы узнаем, принадлежит ли слово $f(x)$ языку L_2, мы тем самым получим ответ на вопрос о принадлежности слова x языку L_1.

Лемма 36.3

Если язык $L_1 \subseteq \{0,1\}^*$ сводится за полиномиальное время к языку $L_2 \subseteq \{0,1\}^*$, то то из $L_2 \in \mathrm{P}$ следует $L_1 \in \mathrm{P}$.

Доказательство Пусть A_2 — полиномиальный алгоритм, распознающий язык L_2, а F — полиномиальный алгоритм, сводящий язык L_1 к языку L_2. Построим алгоритм A_1, который будет за полиномиальное время разрешать язык L_1.

Рисунок 36.4 иллюстрирует построение. Получив вход $x \in \{0,1\}^*$, алгоритм A_1 (с помощью алгоритма F) получает $f(x)$ и с помощью алгоритма A_2 проверяет, принадлежит ли слово $f(x)$ языку L_2. Результат работы алгоритма A_2 на слове $f(x)$ и выдается алгоритмом A_1 в качестве ответа.

Определение (36.1) гарантирует, что алгоритм A_1 даёт правильный ответ; он полиномиален, поскольку полиномиальны алгоритмы F и A_1 (упр. 36.1-6).

36.5
Предполагаемое соотношение между классами P. NP и NPC. Классы P и NPC содержатся в NP (что очевидно), и, можно полагать, не пересекаются и не покрывают всего NP.

NP-полнота

Понятие сводимости позволяет придать точный смысл утверждению о том, что один язык не менее труден, чем другой (с точностью до полинома). Запись $L_1 \leqslant_\mathrm{P} L_2$ можно интерпретировать так: сложность языка L_1 не более чем полиномиально превосходит сложность языка L_2. Наиболее трудны в этом смысле NP-полные задачи.

Язык $L \subseteq \{0,1\}^*$ называется NP-полным *(NP-complete), если*

1. $L \in \mathrm{NP}$.
2. $L' \leqslant_\mathrm{P} L$ *для любого* $L' \in \mathrm{NP}$

Класс NP*-полных языков будем обозначать* NPC. *Языки, которые обладают свойством 2 (но не обязательно обладают свойством 1), называют* NP-трудными *(NP-hard)*

Основное свойство NP-полных языков состоит в следующем:

Теорема 36.4

Если некоторая NP-полная задача разрешима за полиномиальное время, то $\mathrm{P} = \mathrm{NP}$.

Если в классе NP существует задача, не разрешимая за полиномиальное время, то все NP-полные задачи таковы.

Доказательство

Пусть L — NP-полный язык, который одновременно оказался разрешимым за полиномиальное время ($L \in \mathrm{P}$ и $L \in \mathrm{NPC}$). Тогда для любого $L' \in \mathrm{NP}$ по свойству 2 определения NP*-полного языка имеем $L' \leqslant_\mathrm{P} L$. Следовательно, $L' \in \mathrm{P}$ (лемма 36.3), и первое утверждение теоремы доказано.*

Второе утверждение теоремы является переформулировкой первого.

Таким образом, гипотеза $\mathrm{P} \neq \mathrm{NP}$ *означает, что NP-полные задачи не могут быть решены за полиномиальное время. Большинство экспертов полагает, что это действительно так; предполагаемое соотношение между классами* P, NP *и* CNP *показано на рис. 36.5.*

Конечно, мы не можем быть уверены, что однажды кто-нибудь не предъявит полиномиальный алгоритм для решения NP-полной задачи и не докажет тем самым, что $\mathrm{P} = \mathrm{NP}$. *Но пока что этого никому не удалось, и потому доказательство* NP*-полноты некоторой задачи является убедительным аргументом в пользу того, что она является практически неразрешимой.*

Задача о выполнимости для схем

Как мы уже говорили, мы начнём с одной конкретной задачи; установив её NP-полноту, можно доказывать NP-полноту дру-

ВНИМАНИЕ: внутри треугольников с кружочками надо написать НЕ, внутри гнутых треугольников — ИЛИ, внутри кривых прямоугольников — И.

Два набора входных данных для задачи CIRCUIT-SAT. (a) Набор входных значений $\langle x_1 = 1, x_2 = 1, x_3 = 0\rangle$ даёт значение 1 на выходе, так что эта схема выполнима. (b) Для этой схемы никакой набор входных значений не приводит к единице на выходе, так что эта схема не выполнима.

гих задач методом сведения. В качестве такой задачи мы рассмотрим задачу о выполнимости для схем (circuit-satisfyability problem, или CIRCUIT-SAT). К сожалению, подробное доказательство NP-полноты этой задачи требует рассмотрения технических деталей, выходящего за рамки данной книги. Поэтому мы ограничимся неформальным наброском доказательства, полагая, что читатель имеет представление о схемах из функциональных элементов (см. гл. 29).

На рисунке 36.6 показаны две схемы из функциональных элементов. Обе имеют по три входа и по одному выходу.

Будем рассматривать наборы значений *булевых переменных, соответствующих входам схем (truth assignments). Схема из функциональных элементов с одним выходом называется* выполнимой *(satisfiable), если существует* выполняющий набор *(satisfying assignment), то есть такой набор значений входов, при котором на выходе схемы появляется единица. Например, схема рис. 36.6(а) имеет выполняющий набор* $\langle x_1 = 1, x_2 = 1, x_3 = 0\rangle$ *и потому является выполнимой. В то же время никакие значения переменных* x_1, x_2, x_3 *для схемы рис. 36.6(б) не приводят к появлению* 1 *на выходе. Следовательно, эта схема невыполнима.*

Задача о выполнимости схемы *(circuit-satisfiability problem) требует выяснить, "является ли данная схема, составленная из элементов И. ИЛИ и НЕ, выполнимой". Конечно, нужно договориться о способе представления булевых схем с помощью строк битов — это делается естественным образом (подобно тому, как это делается для графов); при этом размер полученной строки битов не более чем полиномиально зависит от размера схемы. Зафиксировав такое представление, рассмотрим язык*

CIRCUIT-SAT$= \{\langle C\rangle : C$ *— выполнимая схема из функциональных элементов*$\}$

Задачу о выполнимости можно сформулировать так: можно ли данную схему заменить на эквивалентную, в которой выход соединён напрямую с нулевым проводом.

Разумеется, можно узнать, выполнима ли данная схема, перебрав все возможные комбинации значений входов. К сожалению, их много: для схемы с k *входами придется перебрать* 2^k *наборов — время работы такого переборного алгоритма не ограничено полиномом (от* k*, или, что то же, от размера схемы). Как мы уже* *отмечали, большинство специалистов уверены, что что задача о выполнимости схемы не может быть решена полиномиальным алгоритмом (поскольку NP-полна).*

Для начала убедимся, что задача CIRCUIT-SAT принадлежит классу NP.

Лемма 36.5

Задача CIRCUIT-SAT принадлежит классу NP.

Доказательство

Сертификатом является набор входных значений, при которых выходное значение равно 1 *— ясно, что этот факт легко проверить за полиномиальное время.*

Для доказательства NP-полноты задачи CIRCUIT-SAT нам осталось доказать, что данная задача NP-трудна, то есть что любая задача из класса NP сводится к задаче CIRCUIT-SAT за полиномиальное время. Полное доказательство этого утверждения довольно громоздко, так что мы дадим лишь набросок доказательства.

Коротко напомним, как устроен компьютер. Программа хранится в памяти компьютера в виде последовательности инструкций (команд). Инструкции содержат код операции, которую нужно выполнить, адреса аргументов (операндов) команды и адрес, по которому следует записать результат операции. В специальном месте памяти, которое называется счётчиком команд *(program counter), хранится адрес текущей команды. Этот адрес в процессе выполнения программы меняется (постепенно увеличиваясь при выполнении обычных команд или перескакивая в другое место программы в условных операторах и циклах).*

Текущее состояние программы, определяющее дальнейший ход её исполнения, записано в каждый момент в памяти (мы включаем в это понятие регистры процессора, счётчик команд и т.п.). Каждое состояние памяти компьютера будем называть конфигурацией *(configuration). Выполнение инструкций может рассматриваться как последовательное преобразование текущей конфигурации в следующую за ней в соответствии с некоторыми правилами. Эти правила определяются электроникой компьютера и могут быть представлены в виде схемы из функциональных элементов (если конфигурация представляется набором n битов, то эта схема имеет n входов и n выходов). В доказательстве следующей леммы мы будем обозначать эту схему через M.*

Лемма 36.6

Задача CURCUIT-SAT является NP-трудной.

Доказательство

Пусть L — произвольный язык из класса NP. Мы построим функцию f, которая будет отображать каждое двоичное слово в схему $C = f(x)$, для которой $x \in L$ равносильно

переводы надписей:
input bits — сертификат aux machine state — регистры процессора
working storage — рабочая память
0/1 output — выходной бит

36.7 Последовательность конфигураций, соответствующая работе алгоритма A для входа x и сертификата y. Каждая конфигурация соответствует состоянию вычисления в какой-то момент времени и включает в себя (помимо A, x и y) также значение счётчика команд (PC), а также содержимое регистров процессора и рабочей памяти. Каждая конфигурация переводится в следующую с помощью схемы M из функциональных элементов. Выходной бит записан в выделенном месте рабочей памяти.

$C \in \mathrm{CIRCUIT-SAT}$. *(Функция f будет вычисляться полиномиальным алгоритмом.)*

Поскольку $L \in \mathrm{NP}$, существует алгоритм A, проверяющий данный язык за полиномиальное время. Мы используем этот алгоритм для построения сводящего алгоритма F. Пусть $T(n)$ — наибольшее время работы алгоритма A на входах длины n (напомним, что мы называем входом первый аргумент алгоритма A; второй называется сертификатом). Выберем такое число k, что $T(n) = O(n^k)$ и длина сертификатов для входов длины n тоже есть $O(n^k)$. (Время работы алгоритма полиномиально зависит от общей длины условия и сертификата. Но поскольку длина сертификата ограничена полиномом от длины условия, время работы алгоритма также ограничено полиномом от n.)

Идея доказательства состоит в представлении работы алгоритма A в виде последовательности конфигураций. Как показано на рис. 36.7, каждая конфигурация состоит из части, содержащей программу, счётчика команд, состояния регистров процессора, входа x, сертификата y и рабочей памяти.

Начальное состояние памяти компьютера образует конфигурацию c_0. Затем каждая конфигурация c_i преобразуется в следующую конфигурацию c_{i+1}, причём это преобразование выполняется с помощью схемы из функциональных элементов M, реализованной в электронике компьютера. В конце работы алгоритм получает ответ (ноль или единицу). Будем считать, что этот ответ записывается в определённое место памяти. После получения ответа программа останавливается, и состояние памяти в дальнейшем не изменяется. Следовательно, если алгоритм делает не более $T(n)$ шагов, результат работы алгоритма находится в фиксированном бите конфигурации $c_{T(n)}$.

Таким образом, можно построить схему, которая последовательно вычисляет конфигурации $c_1, c_2, \ldots, c_{T(N)}$. Эта схема состоит из последовательно соединённых $T(n)$ копий схемы M. Выход i-ой копии схемы M будет конфигурацией c_i. Выходными зна-

чениями схемы будут биты конфигурации $c_{T(n)}$*. Описанную схему назовем* C'*.*

Вспомним, что должен делать сводящий алгоритм F*. Получив входное слово* x*, он должен построить схему* $C = f(x)$*, которая выполнима, если и только если существует такой сертификат* y*, что* $A(x, y) = 1$*. Для этого найдём длину* n *слова* x *и построим схему* C' *описанным способом. Входом схемы* C' *будет начальная конфигурация вычисления* $A(x, y)$*, а выходом — конфигурация* $c_{T(n)}$*.*

Затем полученную схему C' *следует немного преобразовать. Во-первых, нужно установить значения входных переменных, соответствующих программе* A*, начальному положению счетчика команд, входу* x*, исходному состоянию служебной информации и рабочей части памяти. (Для этого мы соединяем соответствующие входы схемы с постоянными сигналами* 0 *или* 1*.) Неопределёнными остаются только те входные переменные, которые отвечают за значение сертификата* y*. Во-вторых, игнорируются все выходы схемы* C'*, кроме одного — того бита конфигурации* $c_{T(n)}$*, в котором хранится результат работы алгоритма* A*. Новая, преобразованная схема и есть нужная нам схема* C*. Сводящий алгоритм, получив вход* x*, строит схему* C *и выдаёт её (точнее, её представление в виде строки битов).*

Остается доказать, что сводящий алгоритм обладает двумя необходимыми свойствами. Во-первых, нужно показать, что F *корректно вычисляет сводящую функцию* f*. Это значит, что схема* C *выполнима тогда и только тогда, когда существует сертификат* y*, для которого* $A(x, y) = 1$*. Во-вторых, нужно показать, что алгоритм* F *работает полиномиальное время.*

Докажем корректность алгоритма F*. Пусть существует сертификат* y *длины* $O(n^k)$*, для которого* $A(x, y) = 1$*. Подставим биты сертификата* y *во входные переменные схемы* C*. Тогда выход схемы* $C(y)$ *будет равен* $A(x, y) = 1$*. Таким образом, если существует сертификат, то схема* $C = f(x)$ *выполнима. Наоборот, если схема* C *выполнима, то для некоторого набора* y *значений входных переменных имеем* $C(y) = 1$*, и потому* $A(x, y) = 1$*. Корректность алгоритма* F *доказана.*

Осталось заметить, что размер схемы C *и время работы сводящего алгоритма полиномиально зависят от размера входа* $n = |x|$*. Прежде всего заметим, что каждая конфигурация содержит полиномиальное число битов. Действительно, длина программы* A *фиксирована и не зависит от входа, длина входа* x *равна* n*, а длина сертификата* y *полиномиально зависит от* n*. Поскольку алгоритм* A *делает только полиномиальное число шагов, размер используемой им памяти тоже полиномиален. (Мы считаем, что не только число фактически использованных ячеек памяти, но и общее число ячеек памяти полиномиально. На самом деле это* *ограничение несущественно, см. упр. 36.3-4.) Таким образом, и число уровней на рис. 36.7, и размер каждого уровня полиномиальны; конструкция схемы также регулярна и построение её за полиномиальное время не составляет труда.*

Итак, мы доказали, что язык CIRCUIT-SAT лежит в классе NP и что любой язык из NP к нему сводится, то есть что задача CIRCUIT-SAT является NP-полной.

Теорема 36.7

Задача о выполнимости схемы NP-полна.

Упражнения

36.3-1

Покажите, что отношение $\leqslant_{\mathrm{P}}$ *транзитивно, то есть что из* $L_1 \leqslant_{\mathrm{P}} L_2$ *и* $L_2 \leqslant_{\mathrm{P}} L_3$ *следует* $L_1 \leqslant_{\mathrm{P}} L_3$.

36.3-2

Докажите, что $L \leqslant_{\mathrm{P}} \bar{L}$, *если и только если* $\bar{L} \leqslant_{\mathrm{P}} L$

36.3-3

Проверьте, что в доказательстве леммы 36.5 в качестве сертификата можно взять значения на всех проводах (входных, выходных и внутренних) схемы.

36.3-4

В доказательстве леммы 36.6 мы предполагали, что компьютер использует участок памяти полиномиального размера (а не полиномиальное число ячеек в разных местах памяти экспоненциального размера). Почему это было существенно? Как обойтись без этого предположения?

36.3-5

Язык L называется полным *в классе языков* C *относительно полиномиальной сводимости, если* $L \in C$ *и* $L' \leqslant_p L$ *для любого* $L' \in C$. *Докажите, что в классе* P *все языки (кроме* $\emptyset$ *и* $\{0,1\}^*$*) полны относительно полиномиальной сводимости.*

36.3-6

Покажите, что язык L полон в классе NP *тогда и только тогда, когда язык* $\bar{L}$ *полон в* co − NP.

36.3-7

Сводящий алгоритм F *(лемма 36.6) строит схему* $C = f(x)$, *используя информацию об* x, A *и* k. *Профессор заметил, что алгоритм* F *получает в качестве аргумента только* x, *однако ни* A, *ни* k *ему не даны. (Поскольку язык* L *принадлежит NP, можно утверждать, что требуемые* A *и* k *существуют, но каковы они, мы не знаем.) На этом основании профессор делает вывод, что алгоритм* F *построить нельзя, и что язык CIRCUIT-SAT не обязан принадлежать NP. Объясните, в чём ошибка профессора.*

Доказательства NP-полноты.

Доказывая NP-полноту языка CIRCUIT-SAT, мы проверяли, что всякий язык из класса NP сводится к CIRCUIT-SAT за полиномиальное время. Больше этого (рассматривать произвольный *язык из класса NP) нам делать не придётся — чтобы доказать NP-полноту какого-либо языка, достаточно свести к нему другой язык, NP-полнота которого уже доказана. Таким способом мы докажем NP-полноту двух задач о выполнимости формул, а в разделе 36.5 — NP-полноту ещё нескольких задач.*

Другими словами, NP-полноту языков мы будем доказывать с помощью следующей леммы.

Лемма 36.8

Если для языка L найдется язык $L' \in \mathrm{CNP}$, *для которого* $L' \leqslant_{\mathrm{P}} L$, *то язык L NP-труден. Если, кроме того,* $L \in \mathrm{NP}$, *то* $L \in \mathrm{CNP}$.

Доказательство

Поскольку $L' \in \mathrm{CNP}$, *любой язык* L'' *из класса NP сводится к* L'. *По условию* $L' \leqslant_{\mathrm{P}} L$, *поэтому по свойству транзитивности (упр. 36.3-1)* $L'' \leqslant_p L$. *Таким образом, язык L является* NP-*трудным. Если теперь* $L \in \mathrm{NP}$, *то* $L \in \mathrm{CNP}$.

Другими словами, нам не надо доказывать, что любой NP-язык сводится к интересующему нас — достаточно проверить это для одного NP-полного языка. Получаем такую схему доказательства NP-полноты языка L.

1. Доказываем, что $L \in \mathrm{NP}$.

2. Выбираем какой-либо известный NP-*полный язык* L'.

3. Строим алгоритм, вычисляющий функцию f, которая отображает входы задачи L' *во входы задачи L.*

4. Доказываем, что функция f сводит L' *к L, то есть что* $x \in L'$ *тогда и только тогда, когда* $f(x) \in L$.

5. Доказываем, что вычисляющий функцию f алгоритм работает полиномиальное время.

Можно сказать и так: мы уже пробили брешь в стене, установив NP-полноту одного языка, и теперь можем использовать это для доказательства NP-полноты других языков. Эти языки также можно использовать как эталонные при доказательстве NP-полноты, так что чем больше наша коллекция NP-полных языков, тем легче доказывать NP-полноту новых.

Выполнимость формул

Мы докажем NP-полноту задачи о выполнимости пропозициональных формул. (Именно для этой задачи впервые была доказана NP-полнота.) Вот как она формулируется.

Пропозициональные формулы составлены из

1. булевых переменных $x_1, x_2, \ldots$;

2. булевских операций (пропозициональных связок) $\wedge$ *(конъюнкция, AND, И),* $\vee$ *(дизъюнкция, OR, ИЛИ),* $\neg$ *(отрицание, НЕ, NOT),* $\Rightarrow$ *(импликация, следование),* $\Leftrightarrow$ *(эквивалентность, если и только если);*

3. скобок.

Так же как и для булевых схем, набором значений *(truth assignment) для пропозицональной формулы* φ *будем называть* *набор значений переменных, входящих в эту формулу.* Выполняющим набором *(satisfying assignment) мы называем набор значений, на котором формула принимает значение* 1 *(истинна). Формула называется* выполнимой *(satisfiable), если для неё существует выполняющий набор.* Задача о выполнимости формулы *(formula satisfiability problem) состоит в проверке, является ли заданная (пропозициональная) формула выполнимой. Другими словами,*

$$SAT = \{\langle\varphi\rangle : \varphi \text{ — выполнимая булева формула}\}.$$

Например, формула

$$\varphi = ((x_1 \Rightarrow x_2) \vee \neg((\neg x_1 \Leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

имеет выполняющий набор $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1\rangle$, *так как*

$$\begin{aligned} \varphi &= ((0 \Rightarrow 0) \vee \neg((\neg 0 \Leftrightarrow 1) \vee 1)) \wedge \neg 0 \\ &= (1 \vee \neg(1 \vee 1)) \wedge 1 \\ &= (1 \vee 0) \wedge 1 \\ &= 1, \end{aligned} \tag{36.2}$$

и, следовательно, данная формула φ *принадлежит SAT.*

Выполнимость формулы можно проверить, перебрав все наборы значений переменных. Однако этот переборный алгоритм не является полиномиальным (для формулы с n *переменными есть* 2^n *вариантов). Как показывает следующая теорема, задача SAT является NP-полной и потому для неё вряд ли существует полиномиальный алгоритм.*

Теорема 36.9

Задача о выполнимости формулы NP-полна.

Доказательство

Прежде всего убедимся, что $SAT \in NP$. *В самом деле, сертификатом является выполняющий набор, а проверяющий алгоритм подставляет значения из этого набора на место переменных и вычисляет значение формулы.*

Осталось показать, что задача SAT NP-трудна. Для этого достаточно проверить, что $\text{CIRCUIT-SAT} \leqslant_{\text{P}} SAT$. *Другими словами, мы должны по данной схеме построить формулу, которая будет выполнима в том и только том случае, когда исходная схема была выполнима.*

Можно построить формулу, которая вычисляет ту же функцию, что и заданная схема. Это делается просто: двигаясь от входов схемы к выходу, мы на каждом проводе пишем формулу, ему соответствующую (например, выходу элемента И соответствует формула $\varphi \wedge \psi$, *где* φ *и* ψ *— формулы, соответствующие её входам).*

36.8
Сведение выполнимости схемы к выполнимости формулы. Для каждого провода схемы заводим переменную; формула представляет собой конъюнкцию утверждений о поведении каждого элемента.

К сожалению, этот метод не является полиномиальным. Многократное использование одних и тех же подформул может привести к формуле экспоненциального размера (упр. 36.4-1). Поэтому нужно действовать более аккуратно.

Трюк состоит в том, чтобы увеличить число переменных в формуле, введя дополнительную переменную для каждого провода в схеме, (а не только для её входов). Основная идея показана на рис. 36.8.

Формула, которую строит сводящий алгоритм, есть конъюнкция переменной, соответствующей выходу схемы, и утверждений о корректности работы всех функциональных элементов. Например, схеме на рисунке 36.8 будет поставлена в соответствие формула

$$\begin{aligned}\varphi = x_{10} &\wedge (x_4 \Leftrightarrow \neg x_3)\\ &\wedge (x_5 \Leftrightarrow \neg(x_1 \vee x_2))\\ &\wedge (x_6 \Leftrightarrow \neg x_4)\\ &\wedge (x_7 \Leftrightarrow \neg(x_1 \wedge x_2 \wedge x_3))\\ &\wedge (x_8 \Leftrightarrow \neg(x_5 \vee x_6))\\ &\wedge (x_9 \Leftrightarrow \neg(x_6 \vee x_7))\\ &\wedge (x_{10} \Leftrightarrow \neg(x_7 \wedge x_8 \wedge x_9))\end{aligned}$$

Это построение, как легко видеть, приводит к схеме полиномиального размера и выполняется за полиномиальное время.

Почему схема C выполнима тогда и только тогда, когда выполнима построенная формула φ? Пусть схема C имеет выполняющий набор. Подавая на вход схемы C значения переменных из этого набора, мы получим определенные значения на каждом проводе схемы. При этом выходным значением схемы будет 1. Эти значения и будут выполняющим набором для формулы φ.

Напротив, если мы имеем некоторый выполняющий набор для формулы φ, то он задаёт некоторое согласованное распределение значений по проводам схемы, при котором выходное значение равно 1, и потому схема выполнима.

Таким образом, CIRCUIT-SAT $\leqslant_{\mathrm{P}}$ SAT, что и требовалось доказать.

Выполнимость 3-CNF формул

Для многих задач удаётся доказать их NP-полноту, сведя к ним задачу о выполнимости формул. При этом удобно ограничить класс формул, выполнимость которых нас интересует — чем этот класс меньше, тем легче осуществить сведение. (При этом, конечно, задача о выполнимости формул из этого класса *должна оставаться NP-полной.) На практике оказывается удобным класс формул в 3-конъюнктивной нормальной форме (3-conjunctive normal form), сокращённо обозначаемый 3-CNF. Такие формулы представляют собой конъюнкцию нескольких подформул, каждая из которых есть дизъюнкция ровно трёх различных литералов (литерал — это переменная или отрицание переменной).*

Например, формула

$$(x_1 \vee \neg x_1 \vee x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

принадлежит классу 3-CNF. Первая из трёх ее дизъюнкций есть $(x_1 \vee \neg x_1 \vee x_2)$*; она содержит литералы* $x_1, \neg x_1$ *и* $\neg x_2$*.*

Задача о выполнимости формул из класса 3-CNF, обозначаемая 3-CNF-SAT, является NP-полной со всеми вытекающими последствиями (вряд ли для неё есть полиномиальный алгоритм; её можно использовать для доказательства NP-полноты других задач).

Теорема 36.10

Задача проверки выполнимости формулы из класса 3-CNF NP-полна.

Доказательство

Язык 3-CNF-SAT принадлежит классу NP по тем же причинам, что и язык SAT (теорема 36.9). Чтобы доказать, что задача 3-CNF-SAT является NP-трудной, мы покажем, что SAT $\leqslant_{\mathrm{P}}$ *3-CNF-SAT, и останется сослаться на лемму 36.8.*

Сведение будет использовать ту же идею, что и доказательство теоремы 36.9 (на самом деле можно было бы модифицировать это доказательство так, чтобы получалась формула из класса 3-CNF).

Пусть дана произвольная формула φ*. Построим "дерево разбора" этой формулы; листья этого дерева соответствуют литералам, а внутренние вершины — пропозициональным связкам. На рис. 36.9 показано такое дерево для формулы*

$$\varphi = ((x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2. \qquad (36.3)$$

Мы считаем, что в формуле не встречаются дизъюнкции или конъюнкции более чем двух подформул (это ограничение не имеет значения, так как всегда можно расставить недостающие скобки). Таким образом, каждая внутренняя вершина имеет одного или двух потомков. Полученное дерево разбора можно рассматривать как схему, вычисляющую булеву функцию (соответствующую формуле).

Далее действуем как в теореме 36.9. Введем переменные y_i *для каждой внутренней вершины построенной схемы. Теперь можно* *вместо исходной формулы написать конъюнкцию переменной, соответствующей корню дерева разбора, и подформул, утверждающих корректность операций в каждой вершине. Например, для формулы (36.3) получится формула*

$$\begin{aligned}\varphi' = y_1 \wedge & (y_1 \Leftrightarrow (y_2 \wedge \neg y_2)) \\ \wedge & (y_2 \Leftrightarrow (y_3 \vee y_4)) \\ \wedge & (y_3 \Leftrightarrow (x_1 \rightarrow x_2)) \\ \wedge & (y_4 \Leftrightarrow (\neg y_5)) \\ \wedge & (y_5 \Leftrightarrow (y_6 \vee x_4)) \\ \wedge & (y_6 \Leftrightarrow (\neg x_1 \leftrightarrow x_3)).\end{aligned}$$

Полученная формула φ' является конъюнкцией нескольких подформул, каждая из которых содержит не более 3 переменных — но эти подформулы ещё не являются дизъюнкциями трёх различных литералов.

Что ж, остаётся заменить каждую из этих подформул на конъюнкцию нескольких литералов. Это делается так: составим таблицу значений для этой подформулы (как на рис. 36.10) и посмотрим на строки, в которых подформула ложна (имеет значение 0). Для каждой из этих строк напишем дизъюнкцию трёх литералов, которая будет ложной как раз при этих значениях переменных, а затем напишем конъюнкцию этих дизъюнкций, которая будет эквивалентна рассмотренной подформуле от трёх переменных.

Для формулы из нашего примера таблица истинности показана на рис. 36.10, формула ложна в четырёх случаях. и записывая конъюнкции, исключающие каждый из этих четырёх случаев, получаем формулу

$$(\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2),$$

которая эквивалентна формуле $(y_1 \Leftrightarrow (y_2 \wedge \neg x_2))$.

Осталась ещё одна небольшая трудность — нам надо, чтобы каждая дизъюнкция содержала ровно три переменных. Для этого можно добавить фиктивные переменные — или на этапе построения таблицы истинности, или сейчас: формулу $p \vee q$ можно заменить на $(p \vee q \vee r) \wedge (p \vee q \vee \neg r)$, а формулу p можно заменить на

$$(p \vee q \vee r) \wedge (p \vee q \vee \neg r) \wedge (p \vee \neg q \vee r) \wedge (p \vee \neg q \vee \neg r) \wedge$$

Легко видеть, что полученная в итоге формула выполнима в том и только том случае, когда выполнима исходная формула (на первом шаге это обосновывается как в теореме 36.9, а дальше мы заменяли формулы на эквивалентные). Ясно также, что длина новой формулы ограничена полиномом от длины старой и что все этапы нашего построения выполнимы за полиномиальное время.

Упражнения

36.4-1

Покажите, что обсуждаемый в начале доказательства теоремы 36.9 прямой метод сведения не годится, предъявив схему размера n, которая переводится в формулу экспоненциального (от n) размера.

36.4-2

Постройте формулу из класса 3-CNF, применив к формуле (36.3) алгоритм из доказательства теоремы 36.10.

36.4-3

Профессор утверждает, что он упростил доказательство теоремы 36.10 и придумал способ преобразовать произвольную формулу в логически эквивалентную ей формулу (с теми же переменными), принадлежащую классу 3-CNF. Почему он не прав?

36.4-4

Покажите, что задача определения, является ли данная пропозициональная формула тавтологией, NP*-полна в классе* co – NP. *(Указание: см. упражнение 36.3-6.)*

36.4-5

Покажите, что за полиномиальное время можно определить, является ли данная булева формула в "дизъюнктивной нормальной форме" (дизъюнкция подформул, каждая из которых является конъюнкцией литералов) выполнимой.

36.4-6

Пусть кто-то придумал полиномиальный алгоритм, проверяющий выполнимость любой пропозициональной формулы. Как можно использовать этот алгоритм для отыскания выполняющего набора (для данной выполнимой формулы) за полиномиальное время?

36.4-7

Рассмотрим язык 2-CNF-SAT, который состоит из выполнимых формул в конъюнктивной нормальной форме, в которых каждый конъюнктивный член является дизъюнкцией не более чем двух литералов. Докажите, что 2-CNF-SAT $\in$ P. Постройте как можно более быстрый разрешающий алгоритм. (Указание: Поскольку $(x \vee y)$ эквивалентно выражению $(\neg x \to y)$, можно свести 2-CNF-SAT к некоторой полиномально разрешимой задаче, связанной с ориентированными графами.)

36.4 NP-полные задачи

В настоящее время известно много NP-полных задач, связанные с самыми разными областями математики и информатики: логикой, теорией графов, компьютерными сетями, множествами *и разбиениями, расписаниями, математическим программированием, алгеброй и теорией чисел, играми и головоломками, оптимизацией программ и т.д. и т.п.*

В этом разделе мы докажем NP-полноту нескольких задач о графах и множествах с помощью метода полиномиального сведения.

Связь между этими задачами показана на рис. 36.11; стрелки означают сводимость за полиномиальное время (CIRCUIT-SAT сводится к SAT и т.д.) Доказав (в теореме 36.7) NP-полноту задачи SAT, с помощью этих сведений мы убеждаемся в полноте всех перечисленных на рисунке задач.

36.4.1 Задача о клике

Кликой *(clique) в неориентированном графе $G = (V, E)$ называется подмножество вершин $V' \subseteq V$, каждые две из которых соединены ребром графа. Другими словами, клика — это полный подграф графа G.* Размером *(size) клики называется число содержащихся в ней вершин. Рассмотрим оптимизационную задачу: определить максимальный размер клики в данной графе. Её называют* задачей о клике *(clique problem). Соответствующая задача разрешения формулируется так: даны граф G и число k; требуется установить, есть ли в графе G клика размера k. Формально говоря,*

$$CLIQUE = \{\langle G, k\rangle : \text{в графе } G \text{ есть клика размера } k\}.$$

Как всегда, мы можем перебрать все подмножества размера k в V и проверить, есть ли среди них клика. Для этого требуется $\Omega(k^2 \cdot C_V^k)$ действий (V — число вершин в графе). При любом фиксированном k эта величина полиномиально зависит от размера графа G. Однако в общей постановке задачи k может быть любым числом, не превосходящим $|V|$, и алгоритм не является полиномиальным. Полиномиального алгоритма скорее всего просто нет, поскольку имеет место следующая

Теорема 36.11

Задача CLIQUE является NP-полной.

Доказательство

Сначала убедимся, что $CLIQUE \in NP$. В самом деле, в качестве сертификата можно взять список всех вершин, образующих клику (имея этот список, наличие всех соединительных рёбер можно проверить за полиномиальное время).

Для доказательства NP-трудности задачи CLIQUE покажем, что 3-CNF-SAT $\leqslant_m$ athrmPCLIQUE. На первый взгляд это кажется странным — пропозициональные формулы никак не связаны с графами и кликами — но на самом деле сведéние построить легко

36.12 Граф. соответствующий формуле $\varphi = C_1 \wedge C_2 \wedge C_3$, где $C_1 = x_1 \vee \neg x_2 \vee \neg x_3, C_2 = \neg x_1 \vee x_2 \vee x_3, C_3 = x_1 \vee x_2 \vee x_3$ при сведении 3-CNF-SAT к задаче о клике. Выполняющий набор $\langle x_1 = 0, x_2 = 0, x_3 = 1\rangle$ выполняет C_1 за счёт $\neg x_2$, а C_2 и C_3 за счёт x_3. Соответствующие вершины показаны светло-серым и образуют клику.

Сводящий алгоритм получает формулу из класса 3-CNF, выполнимость которой нужно проверить (т.е. свести к задаче о клике). Пусть дана формула

$$\varphi = C_1 \wedge C_2 \wedge \ldots \wedge C_k,$$

где каждая подформула C_r есть дизъюнкция трех разных литералов l_1^r, l_2^r и l_3^r. Построим граф $G = (V, E)$, который содержит клику размера k, если и только если формула формула φ выполнима.

Для каждой дизъюнкции $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ из формулы φ нарисуем три вершины v_1^r, v_2^r, v_3^r. Таким образом, граф G будет содержать $3k$ вершин. (Забегая вперёд, можно сказать, что будущая клика будет образована истинными членами дизъюнкций.) Пока что опишем рёбра графа: две вершины v_i^r и v_j^s соединены ребром в графе G, если выполнены следующие условия:

v_i^r и v_j^s принадлежат разным тройкам ($r \neq s$);

литералы l_i^r и l_j^s, которые соответствуют данным вершинам, совместны *(are consistent), то есть не являются отрицаниями друг друга.*

Граф G легко построить по формуле φ за полиномиальное время. На рис. 36.12 показан граф, соответствующий формуле

$$\varphi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3).$$

Покажем, что описанное преобразование действительно является сведением. Сначала предположим, что формула φ имеет выполняющий набор. Тогда каждая дизъюнкция C_i содержит хотя бы один истинный литерал; выберем один из таковых (для каждой дизъюнкции). Отметим соответствующие выбранным литералам вершины v_i^r графа G. Мы утверждаем, что k отмеченных вершин образуют клику. Действительно, два отмеченных литерала совместны, так как оба истинны на одном и том же выполняющем наборе (и потому не могут быть отрицаниями друг друга).

Обратно, пусть в графе G есть клика V' размера k. В каждой тройке вершины не соединены ребрами друг с другом, поэтому клика V' содержит ровно по одной вершине из каждой тройки. Рассмотрим соответствующие литералы и объявим их истинными. Совместность литералов гарантирует, что для этого не *придётся объявлять переменную одновременно истинной и ложной. Если после этого значения некоторых переменных еще не определены, выберем их произвольно. Получим набор значений переменных, который будет выполняющим, так как в каждой из дизъюнкций есть хотя бы один истинный член.*

36.4.2 Задача о вершинном покрытии

Множество вершин $V' \subseteq V$ графа $V = (V, E)$ называется вершинным покрытием *(vertex cover) графа, если у любого ребра графа хотя бы один из концов входит в V'. Если считать, что вершина "покрывает" инцидентные ей рёбра, то вершинное покрытие графа G — это множество вершин, которые покрывают все его рёбра.* Размером *(size) вершинного покрытия называется число входящих в него вершин. Например, граф рис. 36.13 (b) имеет вершинное покрытие $\{w, z\}$ (размера 2).*

Задача о вершинном покрытии *(vertex-cover problem) требует указать минимально возможный размер вершинного покрытия для заданного графа. Как обычно, мы перейдём от задачи оптимизации к задаче разрешения и будем спрашивать, имеет ли данный граф вершинное покрытие данного размера. Соответствующий язык таков:*

VERTEX-COVER $= \{\langle G, k\rangle :$ *граф G имеет вершинное покрытие размера k*$\}$

Теорема 36.12

Задача VERTEX-COVER является NP-полной.

Доказательство

Сначала убедимся, что данная задача принадлежит классу NP. В самом деле, в качестве сертификата годится само вершинное покрытие (легко проверить, что оно имеет требуемый размер и что оно действительно является покрытие, просмотрев все рёбра).

Чтобы доказать, что задача VERTEX-COVER является NP-трудной, сведём к ней задачу о клике. В доказательстве используется понятие дополнения графа. Пусть дан неориентированный граф $G = (V, E)$. Его дополнением *(complement) назовём граф $\bar{G} = (V, \bar{E})$, где $\bar{E} = \{(u, v) : (u, v) \notin E\}$. Другими словами, граф $\bar{G}$ имеет те же вершины, что и граф G, а рёбра соединяют те пары вершин, которые не были соединены ребром в графе G. На рис. 36.13 показаны пример графа и его дополнения, появляющиеся при сведении задачи CLIQUE с задаче VERTEX-COVER.*

Сводящий алгоритм получает на вход граф G и число k; вопрос состоит в том, есть ли в графе g клика размера k. Алгоритм строит граф $\bar{G}$ (дополнение графа G) и даёт на выходе пару $\langle \bar{G}, |V| - k\rangle$. Остаётся заметить, что граф $\bar{G}$ имеет вершинное

36.13
Сведение задачи CLIQUE к задаче VERTEX-COVER.
(a) Неориентированный граф $G = (V, E)$ и клика $V' = \{u, v, x, y\}$.
(b) Его дополнение $\bar{G}$, построенное сводящим алгоритмом, имеет вершинное покрытие $V \setminus V' = \{w, z\}$.

покрытие размера $|V| - k$ *тогда и только тогда, когда граф* G *имеет клику размера* k *(возможность выполнить это преобразование за полиномиальное время очевидна).*

В самом деле, если есть клика размера k*, то её дополнение образует вершинное покрытие графа* $\bar{G}$ *(любое ребро графа* $\bar{G}$ *отсутствует в графе* G*, поэтому один из концов этого ребра должен быть вне клики).*

Напротив, если есть вершинное покрытие графа G *размера* $|V| - k$*, то его дополнение является кликой: если какие-то две вершины этого дополнения не связаны ребром в графе* G*, то они были бы связаны ребром в* $\bar{G}$*, и это ребро не было бы покрыто.*

Поскольку задача VERTEX-CLIQUE является NPполной, вряд ли для неё существует эффективный алгоритм. Однако, как мы увидим в разделе 37.1, существует эффективный алгоритм, дающий " приближённое" решение этой задачи — можно найти вершинное покрытие, в котором число вершин не более чем вдвое превосходит минимально возможное.

Таким образом, NP-полнота задачи ещё не означает, что надо отказаться от идеи её решить — возможно, например, что есть полиномиальный алгоритм, который даёт решение, близкое к оптимальному. (В разделе 37 мы вернёмся к этой идее и рассмотрим приближенные алгоритмы для нескольких NP-полных задач.)

36.5.3 Задача о суммах подмножеств

В этом разделе мы рассмотрим ещё одну NP-полную задачу, на этот раз — арифметическую. Пусть даны конечное множество натуральных чисел $S \subset \mathbb{N}$ *и число* $t \in \mathbb{N}$*. В* задаче о суммах подмножеств *требуется выяснить, существует ли такое подмножество* $S' \subseteq S$*, сумма элементов которого равна* t*. Например, если* $S = \{1, 4, 16, 64, 256, 1040, 1093, 1284, 1344\}$ *и* $t = 3754$*, то ответ будет положительным — можно взять* $S' = \{1, 16, 64, 256, 1040, 1093, 1284\}$.

Соответствующий язык таков:

$$\textit{SUBSET-SUM} = \{\langle S, t\rangle : \text{ существует такое подмножество } S' \subseteq S, \text{ что } t = \sum_{s \in S'} s\}.$$

Напомним, что мы записываем числа в двоичной системе (представляя вход задачи в виде битовой строки).

Теорема 36.13

36.14 Сведение задачи о вершинном покрытии к задаче о суммах подмножеств.

(a) Неориентированный граф G. Светло-серые вершины образуют вершинное покрытие $\{v_1, v_3, v_4\}$ размера 3.

(b) Матрица инцидентности этого графа. Светло-серые строки соответствуют вершинам покрытия.

(c) Соответствующий вход задачи о суммах подмножеств. Внутри рамки находится матрица инцидентности, Вершинное покрытие $\{v_1, v_3, v_4\}$ размера $k = 3$ соответствует подмножеству из светло-серых элементов $\{1, 16, 64, 256, 1040, 1093, 1284\}$, сумма которого есть 3754.

Задача SUBSET-SUM является NP-полной.

Доказательство

Очевидно, эта задача принадлежит классу NP (сертификатом можно считать само подмножество S').

Теперь покажем, что VERTEX-COVER $\leqslant_{\mathrm{P}}$ SUBSET-SUM. Сводящий алгоритм преобразует вход $\langle G, k\rangle$ задачи о вершинном покрытии в пару $\langle S, t\rangle$ с таким свойством: в графе G существует вершинное покрытие размера k тогда и только тогда, когда в S найдется подмножество с суммой t.

Будем использовать представление графа G матрицей инцидентности. Пусть $G = (V, E)$ — неориентированный граф; мы считаем, что его вершинами являются числа $0, 1, 2, \ldots, |V| - 1$, а рёбрами — числа $0, 1, 2, \ldots, |E| - 1$. Тогда матрицей инцидентности *(incidence matrix) графа G будет $|V| \times |E|$-матрица B, определяемая так:*

$$b_{ij} = \begin{cases} 1, & \text{\textit{если ребро} } j \text{ \textit{инцидентно вершине} } i, \\ 0, & \text{\textit{в противном случае.}} \end{cases}$$

Например, матрица инцидентности рис. 36.14 (b) соответствует графу рис. 36.14 (a).

Сводящий алгоритм получает на вход матрицу инцидентности B и число k. Требуется построить множество S и число t. Все числа будем записывать в системе счисления по основанию 4.

Множество S будет состоять из чисел двух типов: одни соответствуют вершинам графа, другие — его рёбрам. Каждой вершине $i \in V$ ставится в соответствие число, которое записывается (в системе по основанию 4) и $|E|$ так: сначала идёт единица, а потом i-ая строка матрицы инцидентности $B = (b_{ij})$ (рис. 36.14 (c)), то есть строка, соответствующая этой вершине. Каждому ребру $j \in E$ сопоставляется число y_j, запись которого содержит единственную единицу в позиции, соответствующей ребру i (т.е. число 4^j).

Остается указать число t. Старшие разряды числа t совпадают с записью числа k по основанию 4, а последующие $|E|$ раз-

рядов заполнены двойками. Более точно,

$$t = k \cdot 4^{|E|} + \sum_{j=0}^{|E|-1} 2 \cdot 4^j.$$

Все построенные числа имеют двоичное представление полиномиального размера и строятся за полиномиальное время.

Теперь нужно проверить, что граф G имеет вершинное покрытие размера k тогда и только тогда, когда в S существует подмножество S' с суммой t. Пусть дано вершинное покрытие $V' \subseteq V$ требуемого размера, содержащее вершины $i_1, i_2, \ldots, i_k$. Включим в множество S' числа, соответствующие этим вершинам. Тогда сумма элементов множества S', записанная по основанию 4, будет выглядеть так: в старших разрядах стоит число k, а во всех младших разрядах стоят цифры 1 или 2 (в зависимости от того, оба конца ребра вошли в вершинное покрытие или только один). Взяв те рёбра, у которых только один конец вошёл в вершинное покрытие и добавив соответствующие им числа в множество S', мы превратим единицы в двойки, то есть получим в сумме число t. (На рис. 36.14 (с) потребовалось добавить четыре строки y_0, y_2, y_3, y_4, чтобы обеспечить недостающие двойки в младших разрядах.)

Обратное рассуждение аналогично. Пусть имеется множество S', сумма элементов которого равна t. Это значит, что в младших разрядах суммы стоят двойки. Поскольку строки, соответствующие рёбрам, могут дать в каждом разряде максимум единицу, но никак не двойку, это значит, что недостающая единица приходит от " вершинных" строк. Следовательно, вершины, соответствующие входящим в S' строкам, образуют вершинное покрытие. А старшие разряды гарантируют, что число вершин в этом покрытии равно k.

36.4.3 Задача о гамильтоновом цикле

Вернёмся к задаче о гамильтоновом цикле (HAM-CYCLE; мы говорили о ней в разделе 36.2).

Теорема 36.14

Задача HAM-CYCLE является NP-полной.

Доказательство

Мы уже объясняли, почему эта задача принадлежит классу NP (гамильтонов цикл можно считать сертификатом).

Чтобы доказать NP-полноту задачи HAM-CYCLE, мы покажем, что 3-CNF-SAT $\leqslant_m$ athrmPHAM-CYCLE. Другими словами, мы опишем алгоритм, преобразующий формулу φ из класса 3-CNF с переменными $x_1, x_2, \ldots, x_n$ в граф $G = (V, E)$, который имеет

36.15
(a) Блок A, используемый при сведении задачи 3-SAT к задаче HAM-CYCLE.
(b)–(c) Если блок A входит в граф G и соединяется с ним лишь в угловых вершинах, то любой гамильтонов цикл в графе G проходит через A одним из двух способов.
(d) Символическое изображение блока A

36.16
Блок B, используемый при сведении задачи 3-CNF-SAT к задаче HAM-CYCLE. Путь из вершины b_1 в b_4 не может проходить по всем трём рёбрам b_1–b_2б b_2–b_3 и b_3–b_4, но может проходить по любому собственному подмножеству этого множества, как показано в (a)–(e). Символическое изображение (f) подчёркивает это свойство: по крайней мере один из трёх путей, на которые указывают стрелки, должен войти в гамильтонов путь.

гамильтонов цикл в том и только том случае, если если исходная формула была выполнимой. В нашей конструкции мы будем использовать некоторые блоки — *куски графа, обладающие некоторыми специальными свойствами.*

Первый из используемых блоков (блок A) показан на рис. 36.15 (a). Предположим, что A входит в некоторый граф G, причём лишь вершины a, a', b, b' могут быть соединены с остальными вершинами графа. Тогда гамильтонов цикл в графе G (если таковой существует) должен проходить через вершины $z_1 - -z_4$, и это может происходить лишь двумя способами (рис. 36.15 (b) и (c)). Поэтому можно символически изображать блок A как на рис. 36.15 (d), имея в виду, что он заменяет два ребра a–a' и $b - -b'$ с таким дополнительным условием: гамильтонов цикл должен содержать ровно одно из двух этих рёбер.

Второй используемый блок (B) показан на рис. 36.16. Предположим, что блок B входит в некоторый граф G, причём B может быть связан с остальной частью графа только через вершины b_1, b_2, b_3 и b_4. Можно проверить, что гамильтонов цикл графа G (если он существует) не может проходить одновременно через три ребра $(b_1, b_2), (b_2, b_3)$ и (b_3, b_4), поскольку тогда цикл не смог бы пройти через все вершины B. Однако гамильтонов цикл может проходить через любое собственное подмножество этой тройки ребёр, как показано на рис. 36.16 (a)–(e). (ещё два симметричных варианта получатся, если перевернуть (b) и (e)). Блок B будем символически изображать как на рис. 36.16(f) (стрелки на рисунке означают, что хотя бы один из трёх путей, на которые они указывают, должен войти в в гамильтонов цикл).

Теперь у нас всё готово для построения графа G, соответствующего формуле из класса 3-CNF. Этот граф G будет состоять

36.17 Граф G, построенный по формуле $\varphi = (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$. Для этой формулы имеется выполняющий набор $x_1 = 0$, $x_2 = 1$, $x_3 = 1$. Соответствующий гамильтонов цикл показан серым. Отметим, что если в наборе $x_m = 1$, то в цикл входит ребро e_m, а если $x_m = 0$, то $\bar{e}_m$.

из нескольких блоков типа A и B (рис. 36.17).

Предположим, что формула φ составлена из k дизъюнкций $C_1, C_2, \ldots, C_k$, каждая из которых содержит ровно 3 литерала. Для каждой дизъюнкции C_i изготовим блок типа B; обозначим через $b_{i,j}$ соответствующие копии вершин b_j. Соединим вершины $b_{i,4}$ и $b_{i+1,1}$ для $i = 1, 2, \ldots, k-1$.

Далее, каждой переменной x_m формулы φ мы сопоставим пару вершин x'_m, x''_m. Эти вершины мы соединим двумя рёбрами e_m и e'_m. (На самом деле, как мы увидим, это будут не рёбра, а более сложные конструкции, так что кратных рёбер не будет.) Идея здесь в том, что гамильтонов цикл будет проходить через ребро e_m, если в выполняющем наборе переменная x_m принимает значение 1, и через e'_m, если эта переменная принимает значение 0.

Чтобы дать гамильтонову циклу возможность пройти по e_m или e'_m, добавим в граф рёбра (x''_m, x'_{m+1}) для $m = 1, 2, \ldots, m-1$. а также ещё два ребра: $(b_{1,1}, x'_1)$ и $(b_{k,4}, x''_n)$ (верхнее и нижнее рёбра, рис. 36.17).

Построение графа ещё не окончено, мы должны связать блоки графа, соответствующие переменным формулы, с блоками, соответствующими её дизъюнкциям. Если j-ый литерал дизъюнкции C_i есть x_m, соединим ребро $(b_{i,j}, b_{i,j+1})$ с ребром e_m с помощью A-блока. Если же j-ым литералом дизъюнкции C_i является $\neg x_m$, мы соединим с помощью A-блока рёбра $(b_{i,j}, b_{i,j+1})$ и $\bar{e}_m$. Так, в примере рис. 36.17 имеем $C_2 = x_1 \vee \neg x_2 \vee x_3$, поэтому мы размещаем три A-блока между рёбрами

$(b_{2,1}, b_{2,2})$ и e_1;
$(b_{2,2}, b_{2,3})$ и $\bar{e}_2$;
$(b_{2,3}, b_{2,4})$ и e_3.

Отметим, что слова "соединить два ребра с помощью A" означают, что каждое из них заменяется на цепочку из пяти новых ребёр и добавляются связывающие их рёбра и вершины, как это предусмотрено конструкцией A-блока (рис. 36.15).

Один литерал l_m может встречаться в нескольких дизъюнкциях (например, $\neg x_3$ на рис. 36.17). В этих случаях требуется "подключить" к соответствующему ребру несколько A-блоков; это можно сделать, если входящие в A-блоки цепочки из пяти рёбер соединить последовательно, как показано на рис. 36.18.

Мы утверждаем, что формула φ выполнима, если и только если построенный граф G имеет гамильтонов цикл. Предположим сна-

36.18

Подробности конструкции для случая, когда ребро e_m (или $\bar{e}_m$) входит в несколько A-блоков.

(a) Фрагмент рис. 36.17.

(b) Граф, который этот фрагмент символизирует.

чала, что в графе G есть гамильтонов цикл h, и докажем выполнимость формулы φ.

Цикл h должен быть устроен так:

сначала проходим ребро $(b_{1,1}, x'_1)$ (будем считать, что слева направо);

затем для каждого m проходим по одному (и только одному) из рёбер e_m, e'_m;

проходим по ребру $(b_{k,4}, x''_n)$ справа налево.

проходим все B-блоки снизу вверх

Заметим, что в действительности мы проходим не по самим рёбрам e_m и $\bar{e}_m$, а по A-блокам, которые к ним привешены (если таковые есть). Отметим также, что если ни к e_m, ни к $\bar{e}_m$ не привешено ни одного A-блока, то переменная x_m не входит в формулу и её можно вообще удалить, так что кратных рёбер действительно нет.

Теперь можно указать выполняющий набор для формулы φ: если ребро e_m принадлежит гамильтонову циклу h, положим $x_m = 1$. В противном случае цикл h содержит ребро $\bar{e}_m$, и мы полагаем $x_m = 0$.

Докажем, что построенный набор является выполняющим для формулы φ. Рассмотрим какую-то дизъюнкцию C_i и соответствующий ей B-блок. Каждое ребро $(b_{i,j}, b_{i,j+1})$ связано с помощью A-блока либо с ребром e_m, либо с ребром $\bar{e}_m$ (в зависимости от того, является ли j-м литералов в дизъюнкции C_i переменная x_m или её отрицание $\neg x_m$). Цикл h проходит через ребро $(b_{i,j}, b_{i,j+1})$, если и только если соответствующий литерал равен 0. Вспомним, что h не может проходить через все три ребра $(b_{i,1}, b_{i,2})$, $(b_{i,2}, b_{i,3})$ и $(b_{i,3}, b_{i,4})$ в B-блоке, поэтому хотя бы одному из этих ребер соответствует истинный литерал дизъюнкции, и дизъюнкция C_i истинна. Это можно сказать про любую дизъюнкцию C_i, так что все они истинны и формула φ истинна для построенного набора значений переменных.

Обратно, пусть φ истинная для некоторого набора значений переменных. Построим цикл h по описанным выше правилам (цикл содержит ребро e_m при $x_m = 1$ и ребро $\bar{e}_m$ при $x_m = 0$; цикл проходит через ребро $(b_{i,j}, b_{i,j+1})$, если и только если j-й литерал дизъюнкции C_i равен 0 на данном наборе. Очевидно, описанные правила позволяют построить гамильтонов цикл по выполняющему набору.

Отметим, наконец, что граф G имеет полиномиальный размер (т.е. его размер ограничен полиномом от размера формулы φ), и что построение графа G по формуле φ проводится за полиномиальное время. Таким образом, задача 3-CNF-SAT сводится за полиномиальное время к задаче HAM-CYCLE, что и требовалось доказать.

36.4.4 Задача коммивояжёра

С задачей о гамильтоновом цикле тесно связана задача коммивояжёра *(traveling salesman problem, TSP). В этой задаче требуется найти оптимальный маршрут посещения n городов.*

Коммивояжёр хочет объехать все города, побывав в каждом ровно по одному разу и вернуться в город, из которого начато путешествие. Известно, что перелёт из города i в город j стоит $c(i, j)$ рублей (считаем цены целыми числами). В терминах теории графов задачу можно сформулировать так: требуется найти в данном графе гамильтонов цикл с наименьшей стоимостью (стоимость цикла есть сумма стоимостей всех его рёбер). Соответствующий язык формально определяется так:

TSP $= \{\langle G, c, k\rangle : G = (V, E)$ *— полный граф,*$c : V \times V \to \mathbb{Z}$ *— функция стоимости,* $k \in \mathbb{Z}$ *и в* G *есть гамильтонов цикл стоимости не более* $k\}$.

Теорема 36.15

Задача коммивояжера является NP-полной.

Доказательство

Очевидно, задача TSP принадлежит классу NP (в качестве сертификата можно взять гамильтонов цикл стоимости не выше k).

Чтобы убедиться, что задача коммивояжёра является NP-трудной, сведём к ней задачу HAM-CYCLE. Чтобы узнать, есть ли в графе G гамильтонов цикл, построим полный граф G' с теми же вершинами; рёбра из G будут иметь цену 0, *а все остальные рёбра — цену* 1. *Очевидно, что в графе G' существует путь коммивояжёра стоимости* 0 *в том и только в том случае, когда граф G имел гамильтонов цикл.*

Упражнения

36.5-1

Задача изоморфизма с подграфом *subgraph-isomorphism problem требует выяснить для пары графов G_1 и G_2, изоморфен ли граф G_1 некоторому подграфу графа G_2. Докажите, что эта задача NP-полна.*

36.5-2

Дана целочисленная матрица A размера $m \times n$ и m-мерный вектор b. Задача 0-1 целочисленного линейного программирования *(0-1 integer-progeamming problem) требует выяснить, существует ли* *такой n-мерный вектор x с элементами из множества $\{0, 1\}$, что $Ax \leqslant b$. Докажите, что данная задача NP-полна. (Указание: сведите к ней задачу 3-CNF-SAT.)*

36.5-3

Докажите, что задача о сумме подмножества становится полиномиальной, если величину требуемой суммы (t) записывать в унарной системе счисления (как последовательность из t единиц).

36.5-4

Задача о разбиении на равные части *(set-partition problem) состоит в следующем: дано множество целых чисел S; выяснить, можно ли разбить его на две части с равными суммами, то есть найти множество $A \subseteq S$, для которого $\sum_{x \in A} x = \sum_{x \in S \setminus A} x$. Покажите, что эта задача является NP-полной.*

36.5-5

Докажите NP-полноту задачи о гамильтоновом пути (упр. 36.2-6).

36.5-6

Задача о самом длинном простом цикле *состоит в отыскании в данном графе простого (без повторяющихся вершин) цикла наибольшей длины. Сформулируйте соответствующую задачу разрешения и докажите её NP-полноту.*

36.5-7

Профессор утверждает, что конструкцию A-блока в доказательстве теоремы 36.14 можно упростить, исключив вершины z_3 и z_4, а также вершины под над ними. Прав ли он — или такое упрощение создаст не предвиденные им проблемы?

Задачи

36-1 Независимое множество

Множество вершин $V' \subseteq V$ графа $G = (V, E)$ называется независимым *(independent), если никакие две его вершины не соединены ребром.* Задача о независимом множестве *(independent-set problem) состоит в отыскании в данном графе независимого множества максимального размера.*

a. Сформулируйте соответствующую задачу разрешения и докажите ее NP-полноту.

b. Предположим, мы имеем "чёрный ящик", с помощью которого можем решать задачу из пункта (a) за единичное время. Как с его помощью находить независимое множество максимального размера (а не только этот размер) за полиномиальное (от $|V|$ и $|E|$) время?

Хотя задача о независимом множестве в общей постановке NP-полна, некоторые её частные случаи могут быть решены за полиномиальное время.

c. Постройте полиномиальный алгоритм, решающий задачу о независимом множестве для графов степени 2. Докажите правильность вашего алгоритма и оцените время его работы.

d. Постройте полиномиальный алгоритм, решающий задачу о независимом множестве для двудольных графов. Докажите правильность вашего алгоритма и оцените время его работы. (Указание: используйте результаты раздела 27.3.)

36-2 Раскраска графа

Назовём k-раскраской *(k-coloring) неориентированного графа $G = (V, E)$ функцию $c : V \to \{1, 2, \ldots, k\}$, для которой $c(u) \neq c(v)$ для всех рёбер $(u, v) \in E$. Если считать, что числа $1, 2, \ldots, k$ обозначают k различных цветов, а $c(v)$ есть цвет вершины v, то условие на раскраску состоит в том, что концы любого ребра имеют разные цвета.* Задача о раскраске графа *(graph-coloring problem) состоит в нахождении минимального количества цветов, необходимого для раскраски данного графа G с соблюдением этого условия.*

a. Постройте эффективный алгоритм, находящий 2-раскраску данного графа (если таковая существует).

b. Сформулируйте задачу разрешения, соответствующую задаче о раскраске графа. Докажите, что сформулированная вами задача разрешима за полиномиальное время в том и только том случае, если задача о раскраске графа разрешима за полиномиальное время.

c. Рассмотрим язык 3-COLOR, состоящий из графов, для которых существует 3-раскраска. Покажите, что если язык 3-COLOR является NP-полным, то и задача пункта (b) является NP-полной.

Чтобы установить NP-полноту языка 3-COLOR, сведём к нему язык 3-CNF-SAT. Пусть имеется формула φ из класса 3-CNF, состоящая из m дизъюнкций и содержащая переменные $x_1, x_2, \ldots, x_n$. Построим по ней граф $G = (V, E)$, который можно раскрасить в три цвета в том и только том случае, когда формула φ выполнима.

Для каждой переменной x_i формулы φ мы добавим в граф две вершины (одна будет обозначаться x_i, другая — $\neg x_i$). Кроме того, для каждой дизъюнкции мы добавим по 5 вершин. Наконец, нам понадобятся три специальные вершины, которые мы будем условно называть TRUE, FALSE *и* RED.

Теперь опишем рёбра графа G. Они делятся на два типа, которые мы условно назовём "литеральные" и "дизъюнктивные". Три литеральных ребра соединяют между собой вершины TRUE, FALSE *и* RED *(тем самым гарантируя, что каждая из трёх вершин будет окрашена в свой цвет). Кроме того, для каждой переменной x_i имеется треугольник из литеральных рёбер, включающий в себя вершины x_i, $\neg x_i$ и* RED. *(К дизъюнктивным рёбрам мы ещё вернёмся.)*

d. Рассмотрим произвольную 3-раскраску графа G с описанными литеральными ребрами. Докажите, что из каждой пары вершин

36.20

Блок, соответствующий дизъюнкции $(x \vee y \vee z)$ (задача 36-2).

x_i, $\neg x_i$ *одна покрашена в цвет* $c(\text{TRUE})$, *то есть в тот же цвет, что вершина* TRUE, *а другая — в цвет* $c(\text{FALSE})$. *Покажите, что есть естественное соответствие между* 3-*раскрасками такого графа и наборами значений переменных.*

Остается описать дизъюнктивные рёбра; они нужны, чтобы наложить на 3-*раскраску условия, соответствующие истинности дизъюнкций, составляющих формулу* φ. *На рис. 36.20 показаны рёбра, соответствующие дизъюнкции* $(x \vee y \vee z)$. *Блок, соответствующий каждой дизъюнкции, состоит из* 3 *вершин для входящих в неё литералов,* 5 *вспомогательных вершин и вершины* TRUE. *Перечисленные вершины соединены рёбрами как на рис. 36.20.*

e. Докажите, что если в в таком блоке литералы (вершины x, y, z*) имеют цвета* $c(\text{TRUE})$ *или* $c(\text{FALSE})$, *то корректная* 3-*раскраска пяти вспомогательных вершин возможна в том и только том случае, когда хотя бы один из литералов имеет цвет* $c(\text{TRUE})$.

f. Завершите доказательство NP-полноты задачи 3-*COLOR.*

Замечания

Прекрасным введением в теорию NP-полноты является книга Гэри и Джонсона [79], содержащая длинный список NP-полных задач из самых разных областей (см. перечисление областей в начале раздела 36.5). Подробное обсуждение NP-полноты и смежных разделов теории сложности вычислений можно найти в книгах Хопкрофта и Ульмана [104] и Льюиса и Пападимитриу [139]. Ахо, Хопкрофт и Ульман [4] также рассматривают NP-полные задачи и сводимость за полиномиальное время (в частности, там рассматривается сведение задачи о вершинном покрытии к задаче о гамильтоновом цикле)

Класс P был определён в 1964 году Кобэмом [44] и независимо в 1965 году Эдмондсом [61]. Эдмондс определил также класс NP и высказал гипотезу $\mathrm{P} \neq \mathrm{NP}$. *В 1971 году Кук [49] ввёл понятие* NP-*полноты и доказал, что задач о выполнимости формулы, а также задача* 3-*CNF-SAT, являются NP-полными. Независимо определение NP-полноты было дано Левиным, который доказал NP-полноту нескольких задач [138]. Карп в 1972 году предложил метод сведения за полиномиальное время и использовал его для доказательства NP-полноты многих задач [116]. Эта статья Карпа содержит исторически первые доказательства NP-полноты задач о клике, вершинном покрытии и гамильтоновом цикле. К настоящему времени благодаря усилиям многих учёных известны сотни NP-полных задач.*

Приведённое нами доказательство теоремы 36.14 заимствовано из книги Пападимитриу и Стайглица [154].

37 Приближенные алгоритмы

Часто возникшая на практике NP-полная задача настолько важна, что мы не можем позволить себе уклониться от неё, сославшись на NP-полноту. Конечно, надежд построить полиномиальный алгоритм для такой задачи мало. Однако это ещё не значит, что с ней вообще ничего не сделаешь. Во-первых, может оказаться, что какой-то экспоненциальный алгоритм работает приемлемое время на реальных данных. Во-вторых, можно пытаться найти (за полиномиальное время — в худшем случае или в среднем) не оптимальное решение, а некоторое приближение к нему. На практике такое близкое к оптимальному решение может быть вполне достаточным. Алгоритмы, дающие такие решения, называют приближёнными алгоритмами *(approximation algorithms). В этой главе мы разберём приближённые алгоритмы для нескольких NP-полных задач.*

Оценки качества приближённых алгоритмов.

Пусть мы решаем оптимизационную задачу, то есть ищем объект с наибольшей или наименьшей стоимостью среди множества объектов, на которых задана функция стоимости. Стоимость любого объекта положительна. Мы говорим, что некоторый алгоритм решает такую задачу с ошибкой не более чем в $\rho(n)$ раз (has a ratio bound $\rho(n)$), если стоимость найденного им решения (обозначим ее C) отличается от стоимости оптимального (которую мы обозначим C^*) не более чем в $\rho(n)$ раз. Формально это условие записывается так:

$$\max(C/C^*, C^*/C) \leqslant \rho(n). \tag{37.1}$$

Эта запись годится для задач на минимум и на максимум. Если мы ищем максимум, то $0 < C \leqslant C^*$, и потому отношение $C/C*$ не превосходит 1, а отношение C^*/C показывает, во сколько раз оптимальное решение больше (=лучше) нашего. Для задач на минимум, напротив, $0 < C^* \leqslant C$, и отношение C/C^* показывает, во сколько раз стоимость нашего решения больше стоимости оптимального. Мы предполагаем, что все стоимости положительны, и поэтому дроби имеют смысл. Заметим, что $\rho(n)$ не мо-

жет быть меньше 1, *так как взаимно обратные величины* C/C^* *и* C^*/C *не могут одновременно быть меньше* 1.

Можно сказать, что оптимальный алгоритм — это алгоритм, который решает задачу с ошибкой не более чем в 1 *раз. Чем ближе к единице оценка ошибки, тем ближе алгоритм к оптимальному.*

Иногда удобнее оценивать качество алгоритма, измеряя относительную ошибку *(relative error). Она определяется (для каждого входа алгоритма) как отношение*

$$\frac{|C - C^*|}{C^*},$$

где (как и раньше) C^* *обозначает стоимость оптимального решения, а* C *— стоимость решения, даваемого алгоритмом. Относительная ошибка всегда неотрицательна. Говорят, что приближённый алгоритм имеет относительную ошибку не более* $\varepsilon(n)$ *(has a relative error bound* $\varepsilon(n)$*), если*

$$\frac{|C - C^*|}{C^*} \leqslant \varepsilon(n) \tag{37.2}$$

для любого входа длины n*. Легко проверить, что относительная ошибка* $\varepsilon(n)$ *может быть оценена сверху через функцию* $\rho(n)$*. Именно,*

$$\varepsilon(n) \leqslant \rho(n) - 1. \tag{37.3}$$

В самом деле, для задач на минимум это неравенство превращается в равенство. Для задач на максимум $\varepsilon(n) = (\rho(n) - 1)/\rho(n)$; *чтобы получить (37.3), достаточно вспомнить, что* $\rho(n) \geqslant 1$.

Для многих задач известны приближённые алгоритмы, решающие задачу с ошибкой не более чем в некоторое фиксированное число раз (независимо от длины входа). В этих случаях мы пишем ρ *или* ε*, не указывая аргумента* n*. В других случаях такие алгоритмы неизвестны, и приходиться довольствоваться алгоритмами, в которых оценка ошибки растёт с ростом* n*. Например, такова ситуация в задаче о покрытии множествами, которая рассматривается в разделе 37.3.*

Для некоторых задач можно улучшать качество приближения (уменьшать относительную ошибку) ценой увеличения времени работы. Такова ситуация с задачей о сумме подмножества (см. раздел 37.4). Эта ситуация заслуживает специального определения.

Схемой приближения *(approximation scheme) для данной оптимизационной задачи называется алгоритм, который, помимо условия задачи, получает положительное число* ε*, и даёт решение с относительной ошибкой не более* ε*. Схема приближения называется* полиномиальной *(polynomial-time approximation* *scheme), если для любого фиксированного $\varepsilon > 0$ время её работы не превосходит некоторого полинома от n (размера входа).*

При этом степень и коэффиценты полинома, ограничивающего время работы, могут сколь угодно быстро расти с уменьшением ε. Более сильное ограничение таково: схема приближения называется полностью полиномиальной *(fully polynomial-time approximation scheme), если время работы обраничено некоторым полиномом от n и от $1/\varepsilon$, где n — размер входа, а ε — оценка относительной ошибки.*

Например, время работы может быть ограничено величиной $(1/\varepsilon)^2 n^3$. В этом случае уменьшение ε, скажем, в 3 раза приводит к увеличению времени работы в 9 раз.

План главы

В первых трёх разделах рассматриваются полиномиальные приближённые алгоритмы для трёх NP-полных задач. Последний раздел содержит пример полностью полиномиальной схемы приближения. В разделе 37.1 приводится полиномиальный приближенный алгоритм, решающий задачу о минимальном вершинном покрытии с ошибкой не более чем в 2 раза. В разделе 37.2 рассматривается задача о коммивояжёре. Если расстояния удовлетворяют неравенству треугольника, то имеется приближённый алгоритм, решающий эту задачу с ошибкой не более чем в 2 раза. В общем случае (когда неравенство треугольника не обязательно) такого алгоритма не существует ни для какой фиксированной оценки ошибки (если только P не равно NP). В разделе 37.3 строится жадный приближённый алгоритм для задачи о покрытии множествами. Даваемое им решение отличается от оптимального не более чем логарифмическим (от числа элементов) множителем. Наконец, в разделе 37.4 мы приводим полностью полиномиальную схему приближения для задачи о сумме подмножества.

37.1 Задача о вершинном покрытии

Эта задача описана в разделе 36.5.2. Там же доказана её NP-полнота. Напомним, что вершинным покрытием *(vertex cover) неориентированного граяа $G = (V, E)$ мы называем некоторое семейство его вершин $V' \subseteq V$ с таким свойством: для всякого ребра (u, v) графа G хотя бы один из концов u, v этого ребра содержится в V'. Размером вершинного покрытия считаем количество входящих в него вершин.*

Задача о вершинном покрытии *(vertex-cover problem) состоит в нахождении* оптимального вершинного покрытия *(optimal vertex cover), то есть вершинного покрытия минимального размера. Эта проблема является NP-трудной, поскольку соответству-*

Рисунок 37.1 37.1 Работа алгоритма Approx-Vertex-Cover (a) Исходный граф G с 7 вершинами и 8 рёбрами. (b) На первом шаге выбрано ребро (b, c) (жирное). Его концы b и c (серые) включаются в вершинное покрытие C. Пунктирные рёбра (a, b), (c, e) и (c, d) покрыты и удаляются. (c) Ребро (e, f) добавляется к C. (d) Ребро (d, g) добавляется к C. (e) Результат работы алгоритма, множество C, содержит 6 вершин b, c, d, e, f, g. (f) Оптимальное вершинное покрытие этого графа содержит только три вершины (b, d, e).

ющая проблема разрешения является NP-полной (по теореме 36.12).

Несмотря на это несложно найти вершинное покрытие, которое хуже оптимального не более чем в 2 раза. Это делает алгоритм Approx-Vertex-Cover*:*

```
4 возьмём произвольное ребро $(u, v)$ из $E'$
6 удалим из $E'$ все рёбра, инцидентные $u$ или $v$
```

На рисунке 37.1 приведюн пример работы алгоритма Approx-Vertex-Cover. *Алгоритм строит вершинное покрытие C постепенно. Вначале C пустов (строка 1), а множество E' содержит все рёбра графа (строка 2). Затем в цикле (строки 3–6) мы берём ребро из E', добавляем его концы u и v в C, а из E' изымаем все рёбра, имеющие своим концом u или v. Время работы этого алгоритма есть $O(E)$ (при соответствующем представлении множества E').*

Теорема 37.1. Алгоритм Approx-Vertex-Cover *работает с ошибкой не более чем в 2 раза.*

Доказательство. Прежде всего заметим, что даваемое им множество C является вершинным покрытием. В самом деле, цикл в строках 3–6 продолжается до тех пор, пока множество непокрытых рёбер E' не станет пустым.

Чтобы убедиться, что число вершин в C не более чем вдвое хуже оптимально, посмотрим на множество A рёбер, выбираемых в ходе работы алгоритма. Никакие два из них не имеют общей вершины (после того, как ребро выбрано в строке 4, все имеющие с ним общую вершину удаляются в строке 6). Поэтому общее число вершин в C вдвое больше числа рёбер в A. Заметим теперь, что любое вершинное покрытие (в частности, оптимальное покрытие C^*) содержит хотя бы одну вершину каждого выбранного ребра, и для разных рёбер эти вершины разные. Таким образом, $|A| \leqslant |C^*|$, и, следовательно, $|C| = 2|A| \leqslant 2|C^*|$. Что и требовалось доказать.

Упражнения.

37.1-1. Приведите пример графа, для которого этот алгоритм всегда даёт неоптимальное решение.

37.1-2. Профессор Шипилов предлагает такой алгоритм решения задачи о вершинном покрытии: взять вершину наибольшей степени, включить её в покрытие, удалить все смежные *рёбра, повторить это ещё раз и так далее. Приведите пример графа, для которого этот способ даёт решение, отличающееся от оптимального более чем в 2 раза.*

37.1-3. Постройте жадный алгоритм, который находит оптимальное вершинное покрытие для графа, являющегося деревом, за линейное время.

37.1-4. Как мы видели в доказательстве теоремы 6.12, задача о вершинном покрытии и задача о клике взаимно дополнительны: минимальное вершинное покрытие является дополнением к максимальной клике в дополнительном графе. Можно ли отсюда заключить, что и для задачи о клике имеется приближённый алгоритм с ошибкой не более чем в константу раз? Почему?

37.2 Задача коммивояжёра

Эта задача описана в разделе 36.5.5 и состоит в следующем. Для каждого ребра (u, v) *полного неориентированного графа* $G = (V, E)$ *известна его стоимость* $c(u, v)$. *Необходимо найти гамильтонов цикл минимальной стоимости. Стоимостью цикла (и вообще любого множества рёбер* $A \subseteq E$*) мы считаем сумму стоимостей его рёбер:*

$$c(A) = \sum_{(u,v)\in A} c(u, v).$$

На практике функция стоимости рёбер обычно удовлетворяет неравенству треугольника, *которое говорит, что промежуточная остановка* w *на пути из* u *в* v *увеличивает его стоимость:*

$$c(u, w) \leqslant c(u, v) + c(v, w)$$

для любых трёх вершин u, v, w. *Например, это неравенство выполнено, если вершинами графа являются точки плоскости и стоимостью ребра считается его длина (расстояние между его концами).*

Как показывает упражнение 37.2-1, даже в предположении неравенства треугольника задача коммивояжёра остаётся NP-полной. Тем самым мало шансов найти полиномиальный алгоритм, дающий оптимальный путь, и имеет смысл искать приближённые алгоритмы.

В разделе 37.2.1 мы рассмотрим приближённый алгоритм, решающий эту задачу (в предположении неравенства треугольника) с ошибкой не более чем в 2 раза. Неравенство треугольника существенно: в общем случае алгоритма, решающего её с ошибкой не более чем в C *раз, не существует ни для какого фиксированного* C *(если только* P *не равно* NP*).*

Рисунок 37.2 Работа алгоритма Approx-MST-Tour. (a) Исходные точки лежат в вершинах целочисленной решётки. (Например, вершина f на 1 правее и на 2 выше вершины h.) Стоимость ребра определяется как (евклидово) расстояние между точками. (b) Минимальное покрывающее дерево T, построенное алгоритмом MST-Prim. Вершина a является его корнем. (Алфавитный порядок названий вершин соответствует порядку их добавления в ходе работы алгоритма MST-Prim.) (c) Обход дерева T начинается с его корня a. Отмечая вершины на пути туда и обратно, мы получим последовательность $a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$. Алгоритм обхода дерева в порядке *"вершина–потомки"* пропускает вершины на обратном пути, и остаётся a, b, c, h, d, e, f, g. (d) Соответствующий цикл H является результатом работы алгоритма Approx-MST-TOUR. Его стоимость примерно равна 19,074. (e) Оптимальный цикл стоимости $\approx 14{,}715$.

37.2.1 Задача коммивояжёра (с неравенством треугольника)

Мы можем воспользоваться алгоритмом MST-Prim *отыскания минимального покрывающего дерева из раздела 24.2, переделав его в цикл. Если выполнено неравенство треугольника, то этот цикл не более чем вдвое длиннее оптимального.*

```
Approx-TSP-TOUR
1 считаем произвольную вершину $r \in V[G]$ "корнем"
2 построим минимальное покрывающее дерево $T$ для графа $G$ (с корнем в $r$) с помощью алгоритма MST-Prim$(G, c, r)$
3 пусть $L$ — перечень вершин этого дерева в порядке обхода "вершина–потомки"
4 return гамильтонов цикл, обходящий вершины в порядке $L$
```

Обход вершин дерева описан в разделе 13.1: рекурсивный алгоритм обхода поддерева с корнем в v посещает вершину v, а затем обходит поддеревья, корнями которых являются дети вершины v.

На рисунке 37.2 показана работа алгоритма. Для данной системы точек на плоскости (a) строится минимальное остовное дерево T; вершина a — его корень (b). Определяемый им порядок обхода вершин (c) даёт гамильтонов цикл (d), который является результатом работы алгоритма Approx-MST-Tour. *Он примерно на 23% длиннее оптимального цикла (e).*

Время работы алгоритма Approx-MST-Tour *равно $\Theta(E) = \Theta(V^2)$ поскольку алгоритм* MST-Prim *применяется к полному графу. (См. упражнение 24.2-2.) Осталось проверить, что найденный цикл не более чем вдвое длиннее оптимального. (При этом мы используем неравенство треугольника.)*

Теорема 37.2

Алгоритм Approx-MST-Tour *решает задачу о коммивояжёре с ошибкой не более чем в 2 раза, если выполнено неравенство треугольника.*

Доказательство. Пусть H^* — гамильтонов цикл наименьшей стоимости. Нам надо доказать, что $c(H) \leqslant 2c(H^*)$ для цикла H, *найденного с помощью нашего алгоритма. Удаляя любое ребро из цикла H^*, мы получаем покрывающее дерево. Его стоимость не превосходит $c(H^*)$. Тем более это верно для оптимального покрывающего дерева T:*

$$c(T) \leqslant c(H^*) \tag{37.4}$$

Пусть мы обходим вершины дерева T, посещая каждую вершину дважды — до посещения её потомков и после. Для дерева на рис. 37.2 этот полный обход W *включает в себя вершины*

$$a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$$

При этом каждое ребро дерева проходится дважды, поэтому суммарная стоимость всех рёбер $c(W)$ будет вдвое больше $c(T)$

$$c(W) = 2c(T). \tag{37.5}$$

Из неравенств (37.4) и (37.5) следует, что

$$c(W) = 2c(H^*). \tag{37.5}$$

тем самым стоимость обхода вершин в порядке W не более чем вдвое превосходит оптимальную.

Однако это ещё не все: W не является циклом, так как мы бываем дважды в одной и той же вершине. Ничего страшного — мы можем удалить лишние вершины, при этом стоимость может только уменьшиться. Если в пути $\ldots \to u \to w \to \ldots$ мы удаляем вершину v, остаётся ребро (u, w), длина которого по неравенству треугольника не больше суммы длин удалённых рёбер (u, v) и (v, w). Когда мы удалим из пути W повторно проходимые вершины (оставив только первое вхождение каждой), получится цикл

$$a, b, c, h, d, e, f, g.$$

который как раз и соответствует обходу дерева в порядке "вершина–потомки". Этот цикл H и будет результатом работы алгоритма Approx-TSP-Tour. *При этом*

$$c(H) \leqslant c(W), \tag{37.7}$$

так как H получен из W удалением вершин. Это неравенство вместе с (37.6) завершает доказательство теоремы.

Хотя алгоритм Approx-TSP-Tour *и гарантирует ошибку не более чем в 2 раза, для практических целей он, как правило, недостаточно хорош, и применяются другие алгоритмы. (Соответствующие ссылки даны в конце главы.)*

37.2.2 Общая задача коммивояжёра

Если неравенства треугольника нет, то хорошие приближения к оптимальному циклу найти за полиномиальное время не удаётся (если только P не равно NP).

Теорема 37.3. Пусть $\mathrm{P} \neq \mathrm{NP}$ *и* $\rho \geqslant 1$. *Тогда не существует полиномиального приближённого алгоритма, решающего общую задачу о коммивояжёре с ошибкой не более чем в* ρ *раз.*

Доказательство. Пусть такой алгоритм (назовём его A) существует для некоторого $\rho \geqslant 1$ *(не ограничивая общности, можно считать* ρ *целым). Мы покажем, как использовать A для решения задачи о гамильтоновом цикле, определённой в разделе 36.5.5. Так как эта задача NP-полна (теорема 36.14), существование решающего её полиномиального алгоритма влечёт* $\mathrm{P} = \mathrm{NP}$ *(теорема 36.4).*

Итак, пусть дан граф $G = (V, E)$, *и мы хотим узнать, есть ли в нём гамильтонов цикл, используя алгоритм A. Для этого рассмотрим полный граф* $G' = (V, E')$ *с множеством вершин* V:

$$E' = \{(u, v) : u, v in V \text{ и } u \neq v\}$$

На его рёбрах определим функцию стоимости следующим образом:

$$c(u, v) = \begin{cases} 1, & \text{если } (u, v) \in E, \\ \rho|V| + 1 & \text{в противном случае.} \end{cases}$$

Легко понять, что время построение G' *и* c *не превосходит полинома от* $|V|$ *и* $|E|$.

Теперь посмотрим на задачу коммивояжёра для графа G' *с функцией стоимости* c. *Если исходный граф* G *имел гамильтонов цикл, то этот цикл является решением задачи о коммивояжёре со стоимостью* $|V|$, *поскольку содержит* $|V|$ *рёбер единичной стоимости. Если же гамильтонова цикла не было, то любое решение задачи о коммивояжёре проходит по одному из новых (не входящих в E) рёбер. Новые рёбра до́роги. Даже если новое ребро только одно, то общая стоимость (одно новое и* $|V| - 1$ *старых) будет*

$$(\rho|V| + 1) + (|V| - 1) > \rho|V|;$$

если новых рёбер несколько, она будет ещё больше. Тем самым имеется большой — больше чем в ρ *раз — разрыв между стоимостью оптимального цикла в* G' *для случаев наличия или отсутствия гамильтонова цикла в* G. *В первом случае есть цикл стоимостью* $|V|$, *во втором все циклы имеют стоимость больше* $\rho|V|$.

Что даст приближённый алгоритм A, применённый к графу G'? *Мы предполагаем, что даваемый им ответ не более чем в* ρ *раз* *хуже оптимального. Следовательно, если G имеет гамильтонов цикл, то A должен его выдать. Если же цикла нет, то A выдаёт цикл стоимости более $\rho|V|$. Тем самым можно различить один случай от другого за полиномиальное время.*

Упражнения

37.2-1

Покажите, что в задаче о коммивояжёре можно (за полиномиальное время) модифицировать функцию стоимости так, чтобы она стала удовлетворять неравенству треугольника, и оптимальные циклы при этом не изменились. Почему это не противоречит теореме 37.3, даже если P = NP*?*

37.2-2

Задачу коммивояжёра можно решать с помощью эвристики ближайшей точки *следующим образом. Начнём с тривиального цикла, содержащего одну вершину (её можно выбрать произвольно). Затем будем расширять его так: выбираем самую близкую в циклу вершину v, смотрим, какая вершина u цикла ближе всего к v и вставляем v в цикл сразу после u. Так мы делаем до тех пор, пока цикл не пройдёт через все вершины. Докажите, что полученный таким образом цикл имеет стоимость не более чем вдвое худшую оптимального.*

37.2-3

Задача о гамильтоновом цикле с короткими рёбрами *состоит в отыскании гамильтонова цикла, у которого длина самого длинного ребра была бы как можно меньше. В продположении неравенства треугольника построить полиномиальный приближённый алгоритм, решающий эту задачу с ошибкой не более чем в* 3 *раза. (Указание: Докажите по индукции, что можно обойти все вершины минимального покрывающего дерева по одному разу, взяв его полный обход и пропуская некоторые вершины, при этом не пропуская более двух внутренних вершин подряд.)*

37.2-4

Докажите, что если вершинами графа являются точки на плоскости, а стоимость ребра — расстояние между его концами, то оптимальный цикл в задаче о коммивояжёре является несамопересекающимся многоугольником.

37.2.3 Задача о покрытии множествами

Эта задача обобщает NP-полную задачу о вершинном покрытии (и потому является NP-трудной). Однако подход, использованный в приближенном алгоритме для задачи о вершинном покрытии, здесь не работает. Вместо этого мы рассмотрим жадный алгоритм; даваемое им решение будет хуже оптимального в логарифмическое число раз. С ростом размера задачи каче-

Рисунок 37.3 37.3. Пример исходных данных для задачи о покрытии множествами. Множество X состоит из 12 чёрных точек. Семейство $\mathcal{F}$ состоит из 6 множеств S_1–S_6. Минимальное покрытие имеет размер 3 (множества S_3, S_4, S_5). Жадный алгоритм даёт покрытие размера 4, велючая в него множества S_1, S_4, S_5 и S_3 (в указанном порядке).

ство решения ухудшается, но всё же довольно медленно (логарифм — медленно растущая функция), поэтому такой подход может быть полезен.

Исходным данными задачи о покрытии множествами *(set-covering problem) являются конечное множество X, а также семейство $\mathcal{F}$ его подмножеств. При этом каждый элемент множества X принадлежит хотя бу одному из подмножеств семейства $\mathcal{F}$:*

$$X = \bigcup_{S \in \mathcal{F}} S.$$

Мы ищем минимальное число подмножеств из $\mathcal{F}$, которые вместе покрывают *множество X, то есть семейство mathcalC наименьшей мощности, для которого*

$$X = \bigcup_{S \in \mathcal{C}} S. \tag{37.8}$$

Такое семейство $\mathcal{C}$ мы буде называть покрытием *множества X. (Пример задачи о покрытии приведён на рис. 37.3.)*

Можно представлять себе X как набор навыков, необходимых для выполнения какого-то задания; есть несколько человек, владеющих некоторыми из них. Надо сформировать минимальную группу для выполнения задания, включающую в себя носителей всех необходимых навыков.

Задачу о покрытии множествами можно сформулировать в варианте, требующем ответа да/нет: "существует ли покрытие размера не болтше k" (для любого заданного k). В упражнении 37.3-2 мы попросим вас доказать, что эта задача является NP-полной.

Жадный приближённый алгоритм

Этот алгоритм основан на простой идее: в каждый момент мы выбираем множество, покрывающее максимальное число ещё не покрытых элементов.

```
U <- X
mathcal{C} <- 0
while U \ne \emptyset
  do выбираем S \in \mathcal{F} с наибольшим |S\cap U|
     U <- U - S
     \mathcal{C} <- \mathcal{C} \cup \{S\}
return \mathcal{C}
```

В примере на рис. 37.3 этот алгоритм выбирает множества в таком порядке: S_1, S_4, S_5 и S_3.

У каждый момент работы алгоритма множество U содержит ещё не покрытые элементы, а семейство $\mathcal{C}$ — уже включённые в покрытие подмножества. На шаге 4 производится жадный выбор: в качестве S берётся множество, покрывающее наибольшее число ещё не покрытых элементов (если таких несколько, берём любое). После этого S добавляется к семейству $\mathcal{C}$, а его элементы удаляются из U. В конце концов множество ещё не покрытых элементов (U) пусто, а $\mathcal{C}$ является покрытием множества X.

Видно, что алгоритм Greedy-Set-Cover *полиномиален (время работы оценивается многочленом от $|X|$ и $|F|$): количество повторений цикла не превосходит $\min(|X|,|\mathcal{F}|)$, а каждое повторение легко реализовать за $O(|X|\cdot|\mathcal{F}|)$ операций, так что всего будет $O(|X|\cdot|\mathcal{F}|\cdot\min(|X|,|\mathcal{F}|))$ операций. В упражнении 37.3-3 мы предложим вам реализовать этот алгоритм за линейное время.*

Анализ алгоритма

Теперь мы должны сравнить размер покрытия, даваемого этим алгоритм, с минимально возможным. Нам понадобится обозначение $H(d)$ для суммы первых d членов гармонического ряда (см. раздел 3.1): $H(d)=\sum_{i=1}^{d}1/i$.

Теорема 37.4.

Размер покрытия, даваемого алгоритмом Greedy-Set-Cover, *превосходит минимально возможный не более чем в*

$$H(\max\{|S|:S\in\mathcal{F}\})$$

раз.

Доказательство.

Жадный алгоритм отбирает множества одно за другим, на каждом шаге выбирая то из них, которое покрывает больше всего непокрытых элементов. Будем представлять себе, что на каждом шаге имеется доллар, который поровну распределяется между всеми вновь покрытыми элементами. Таким образом, каждый элемент получает деньги только однажды — на том шаге, когда он впервые попадает в покрытие, и получает тем больше денег, чем меньше элементов оказались в том же положении. Формально говоря, если элемент x входит в множество S_i, выбранное на i-ом шаге работы алгоритма, и не входит в S_k при меньших k, то он получит

$$c_x=\frac{1}{|S_i-(S_1\cup S_2\cup\ldots S_{i-1})|}$$

долларов. [Заметим в скобках, что выгоднее получать деньги как можно позже, так как по мере продвижения алгоритма количе-

ство вновь покрываемых за раз элементов может только уменьшаться — жадный алгоритм не будет "оставлять лакомый кусок на потом"]

Всего будет израсходовано $|\mathcal{C}|$ долларов, по одному на каждый элемент построенного покрытия $\mathcal{C}$. Эти деньги будут каким-то образом распределены между всеми элементами множества X (элемент x получает c_x долларов). Нам надо показать, что оптимальное покрытие (содержащее минимально возможное число множеств) не может быть сильно меньше нашего. Мы сделаем это, убедившись, что для любого множества S из семейства $\mathcal{F}$ общая сумма денег, полученная всеми элементами S, не превосходит $H(|S|)$, и потому понадобится не менее $|\mathcal{C}|/H(\max\{|S| : S \in \mathcal{F}\})$ элементов оптимального покрытия, чтобы набрать общую сумму в $|\mathcal{C}|$.

Формально говоря, для оптимального покрытия $\mathcal{C}^*$ мы имеем

$$|\mathcal{C}| = \sum_{x \in X} c_x \leqslant$$

$$\leqslant \sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x \qquad (37.9)$$

и, кроме того, для любого S из семейства $\mathcal{F}$ мы вскоре докажем, что

$$\sum_{x \in S} c_x \leqslant H(|S|).$$

Из неравенств (37.9) и (37.10) следует, что

$$\mathcal{C} \leqslant \sum_{S \in \mathcal{C}^*} H(|S|) \leqslant$$

$$\leqslant |\mathcal{C}^*| \cdot H(\max\{|S| : S \in \mathcal{F}\}),$$

что и составляет утверждение теоремы.

Осталось доказать неравенство (37.10). Пусть S — произвольное множество из семейства $\mathcal{F}$, содержащее u элементов. Каждый из этих элементов получит какую-то сумму денег по нашим правилам, и надо проверить, что общая сумма для всех элементов S не превосходит суммы

$$1 + 1/2 + 1/3 + \ldots + 1/u$$

(эта сумма также состоит из u слагаемых). Сейчас мы в этом убедимся. Посмотрим на группу элементов S, которые получают свои деньги первыми (среди элементов S). Каждый из них получит не более $1/u$. В самом деле, общее количество элементов X, получающих деньги на этом шаге, не может быть меньше u, ведь жадный алгоритм выбирает множество, покрывающее

наибольшее возможное число ещё не покрытых элементов, и не может выбрать множества, худшего S. Значит, на каждый элемент придется не более $1/u$, и общая сумма будет не больше, чем хвост нашей суммы, имеющий ту же длину. После этого останется какое-то число u_1 непокрытых элементов, и надо будет доказать, что общая сумма денег, им причитающаяся, не превосходит оставшейся части нашей суммы, то есть

$$1 + 1/2 + 1/3 + \ldots + 1/u_1$$

Посмотрим на элементы, которые будут получать деньги первыми (среди оставшихся): по тем же причинам, что и раньше, они получат не более $1/u_1$ каждый, и общая сумма не будет превосходить соответствующей части нашей суммы. Продолжая это рассуждение, мы видим, что если выплаты происходят в k этапов и если

$$u_1 \geqslant u_2 \geqslant \ldots \geqslant u_{k-1} \geqslant u_k = 0$$

— количество элементов в S, которым ещё не выдали денег после $1, 2, \ldots, k$ выплат, то общая сумма выплаченных элементам S денег не превосходит

$$\frac{1}{u_{k-1}}+\ldots+\frac{1}{u_{k-1}}+\frac{1}{u_{k-2}}+\ldots+\frac{1}{u_{k-2}}+\ldots+\frac{1}{u_1}+\ldots+\frac{1}{u_1}+\frac{1}{u}+\ldots+\frac{1}{u},$$

что не превосходит (сравниваем почленно) частной суммы гармонического ряда

$$\frac{1}{1}+\ldots+\frac{1}{u_{k-1}}+\frac{1}{u_{k-1}+1}+\ldots+\frac{1}{u_{k-2}}+\ldots+\frac{1}{u_2+1}+\ldots+\frac{1}{u_1}+\frac{1}{u_1+1}+\ldots+\frac{1}{u},$$

то есть $H(u)$, что завершает доказательство неравенства (37.10) и теоремы 37.4

[Вадик: надо бы набрать соответствующие члены один под другим!!]

Следствие 37.4.

Алгоритм Greedy-Set-Cover даёт решение задачи о покрытии, худшее оптимального не более чем в $(\ln|X| + 1)$ раз.

Доказательство. Очевидное следствие теоремы 37.3 и неравенства (3.12).

Если множества, из которых надо выбирать покрытие, содержат мало элементов, то алгоритм Greedy-Set-Cover даёт решение, довольно близкое к оптимальному. Например, если мы применяем этот алгоритм к задаче о вершинном покрытии графа, в котором степени вершин не превосходят 3, то даваемое им решение не более чем в $H(3) = 11/6$ раз хуже оптимального. (Эта *оценка немного лучше, чем для приведённого в разделе 37.1 алгоритма* Approx-Vertex-Cover.*)*

Упражнения.

37.3-1

Рассмотрим каждое из слов `arid`, `dash`, `drain`, `heard`, `lost`, `nose`, `shun`, `slate`, `snare`, `thread` *как множество букв. Что даст алгоритм* Greedy-Set-Cover *в применении к этим множествам? (Если возникает выбор между несколькими словами, берётся первое в алфавитном порядке.)*

37.3-2

Покажите, что задача о покрытии множествами, рассматриваемая как задача разрешения, является NP-полной, сведя к ней задачу о вершинном покрытии.

37.3-3

Показать, что можно реализовать алгоритм Greedy-Set-Cover *с временем работы* $O(\sum_{S\in\mathcal{F}}|S|)$.

37.3-4

Объясните, почему следующее ослабление утверждения теоремы 37.4 очевидно:

$$|\mathcal{C}| \leqslant |\mathcal{C}^*| \cdot \max\{|S| : S \in \mathcal{F}\}.$$

37.5

Приведите примеры, показывающие, что количество различных ответов, даваемых алгоритмом Greedy-Set-Cover *при разных способах выбора множества в строке 4 (из множеств, покрывающих одинаковое число ещё не покрытых элементов), может экспоненциально расти с ростом размера задачи.*

37.3 Задача о сумме подмножества

Исходным данным для этой задачи является пара (S, t), *где* $S = \{x_1, x_2, \ldots, x_n\}$ *представляет собой некоторое множетсво положительных целых чисел, а* t *— положительное целое число. Зная* S *и* t, *надо выяснить, можно ли найти подмножество множества* S, *сумма элементов которого в точности равна* t. *Эта задача является NP-полной (см. раздел 36.5.3).*

Задачу можно ставить и в оптимизационном варианте, требуя отыскать среди подмножеств, сумма которых не превосходит t, *такое, у которого сумма ближе всего к* t. *Можно представлять себе, что мы должны как можно больше загрузить машину грузоподъёмности* t, *имея ящики весов* $x_1, \ldots, x_n$ *(но не переходя границы).*

В этом разделе мы приводим алгоритм, решающий эту задачу за экспоненциальное время, и показываем, как из него получить полностью полиномиальную схему приближения. (Напомним: это означает, что время работы оценивается многочленом от размера задачи и от $1/\varepsilon$, где ε — относительная ошибка.)

Экспоненциальный алгоритм

Если L — набор чисел, а x — некоторое число, то через $L + x$ мы обозначаем набор чисел, который получится, если добавить x к каждому из элементов L. Например, для $L = \langle 1, 2, 3, 5, 9 \rangle$ и $x = 2$ мы имеем $L + x = \langle 3, 4, 5, 7, 11 \rangle$. Аналогичная запись используется и для множеств:

$$S + x = \{s + x : s \in S\}$$

Нам понадобится процедура Merge-Lists(L, L'), результатом которой является соединение двух упорядоченных наборов L и L' с сохранением порядка. Вспоминая сортировку слиянием (раздел 1.3.1). мы видим, что это можно сделать за время $O(|L| + |L'|)$. (Мы не приводим текста этой процедуры.)

Теперь мы можем написать алгоритм Exact-Subset-Sum, решающий сформулированную выше задачу о сумме подмножества. Исходными данными для него является набор положительных целых чисел $S = \langle x_1, x_2, \ldots, x_n \rangle$ и положительное целое число t. Результатом работы является максимально возможная сумма некоторых элементов из S, не превосходящая t.

```
Exact-Subset-Sum (S,t)
n <- |S|
L_0 <- <0>
for i<-1 1 to n
 do L_i <- Merge-Lists(L_{i-1}, L_{i-1}+x_i)
    удалить из $L_i$ элементы, большие $t$
return наибольший элемент в $L_n$
```

Идея алгоритма проста: если через P_i обозначить множество всех чисел, которые можно получить, складывая некоторые из $x_1, x_2, \ldots, x_i$, то

$$P_i = P_{i-1} \cup (P_{i-1} + x_i) \tag{37.11}$$

(элемент x_i может не входить в сумму, а может и входить) мы видим (индукция по i, упражнение 37.4-1), что L_i представляет собой список элементов множества P_i в порядке возрастания (из которого выброшены элементы, большие t).

Сколько времени требует этот алгоритм? Список L_i может содержать до 2^i элементов, так что алгоритм экспоненциален. Впрочем, если t (или все элементы S) ограничено сверху многочленом от $|S|$, то время работы алгоритма также ограничено многочленом от $|S|$.

Полностью полиномиальная схема приближения

Такая схема получается из описанного алгоритма, если хранить списки L_i в сокращенной форме. Степень сокращения опре-

деояется параметром δ - чем он меньше, тем ближе список к полному. Список L' называется δ-сокращением списка L, если L' является частью L и для любого элемента y из L в списке L' найдётся не превосходащий его элемент z, для которого

$$\frac{y-z}{y} \leqslant \delta$$

или, другими словами,

$$(1-\delta)y \leqslant z \leqslant y.$$

Тем самым любой элемент y, выброшенный из L при получении меньшего списка L', представлен в L' своим приближением снизу с относительной ошибкой (относительно y) не более δ. Например, для $\delta = 0{,}1$ и

$$L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29\rangle$$

список

$$L' = \langle 10, 13, 15, 20, 23, 29\rangle$$

является δ-сокращением L, в котором выброшенное число 11 представлено числом 10, числа 21 и 22 представлены числом 20, а число 24 представлено числом 23. Важно иметь в виду, что δ-сокращение является частью оригинального списка. При этом число элементов может сильно уменьшится, но для всех выброшенны элементов чуть меньшие значения останутся.

Следующая процедура сокращает список $L = \langle y_1, y_2, \ldots, y_m\rangle$, упорядоченный по возрастанию, за время $\Theta(m)$. Её результат L' является δ-сокращеним L; список L' также упорядочен по возрастанию.

```
\textsc{Trim} (L, \delta)
m <- |L|
L' <- <y_1>
last <- y_1
for i<-2 to m
  do if last < (1-\delta) y_i
        then дописать $y_i$ в конец списка L'
             last <- y_i
return L'
```

Элементы списка L просматриваются в порядке возрастания, и в L' помещается первый из них, а также те, которые слишком велики, чтобы быть представленными последним из уже имеющихся элементов списка L'.

Теперь схема приближения для задачи о суммах подмножеств может быть построена следующим образом. Напомним, что исходными данными для неё являются последовательность $S =$

$\langle x_1, x_2, \ldots, x_n\rangle$ *из* n *чисел (идущих в произвольном порядке), число* t*, а также параметр приближения* ε *(в интервале* $0 < \varepsilon < 1$*).*

```
\textsc{Approx-Subset-Sum}(S,t,\varepsilon)
1 n <- |S|
2 L_0 <- <0>
3 for i<-1 to n
4   do L_i <- Merge-Lists(L_{i-1},L_{i-1}+x_i)
5      L_i <- Trim (L_i, \varepsilon/n)
6      все элементы $L_i$, большие $t$, удаляются
7 z <- наибольший элемент $L_n$
8 return z
```

Вначале (строка 2) в список L_0 *помещается единственный элемент* 0*. В строках 3–6 мы последовательно (для* $i = 1, \ldots n$*) вычисляем список* L_i*, который окажется сокращением множества* P_i *(всевозможных сумм, составленных из первых* i *элементов) из которого удалены все элементы, большие* t*. Отметим, что это требует специальной проверки, так как сокращения производятся на каждом шаге, и в строке 4 соединяются уже сокращенные варианты списков.*

Прежде чем проверять это, рассмотрим пример: пусть

$$L = \langle 104, 102, 201, 101\rangle$$

при этом $t = 308$ *и* $\varepsilon = 0{,}2$*. Тогда на каждом шаге производиться* $0{,}05$*-сокращение, и вычисления проходят так:*

```
строка 2: L_0 = <0>,

строка 4: L_1 = <0,104>
строка 5: L_1 = <0,104>
строка 6: L_1 = <0,104>

строка 4: L_2 = <0,102,104,206>
строка 5: L_2 = <0,102,206>
строка 5: L_2 = <0,102,206>

строка 4: L_3 = <0,102,201,206,303,407>
строка 5: L_3 = <0,102,201,303,407>
строка 6: L_3 = <0,102,201,303>

строка 4: L_4 = <0,101,102,201,203,302,303,404>
строка 5: L_4 = <0,101,201,302,404>
строка 6: L_4 = <0,101,201,302>
```

В ответе получается $z = 302$*, что отличается от оптимального варианта* ($307 = 104+102+101$) *менее чем на* 2% *(что много меньше разрешённого отклонения в* 20%*).*

Теорема. Алгоритм APPROX-SUBSET-SUM *представляет собой полностью полиномиальную схему приближения для задачи о сумме подмножества.*

Доказательство. Обе операции (сокращение списка в строке 5 и удаление слишком больших элементов в строке 6) могут лишь уменьшить список, так что список L_i остаётся подмножеством множества P_i. (Напомним, что через P_i обозначается множество всех чисел, которые можно получить, складывая некоторые из $x_1, x_2, \ldots, x_i$.) Таким образом, число z действительно будет суммой некоторого подмножества, причём $z \leqslant t$. Осталось проверить лишь, что оно не меньше $(1-\varepsilon)$, умноженного на максимально возможную сумму, не превосходящую t. (Именно этого требует неравенство (37.2) для нашейго случая.)

Посмотрим, к какой ошибке приводит сокращение списка на каждом шаге. При сокращении остающееся число будет меньше вычеркнутого, но не намного: не более чем на (ε/n)-ую долю. Другими словами, остающийся элемент не меньше вычеркнутого, умноженного на $(1-\varepsilon/n)$. Рассуждая по индукции, легко видеть, что для всякого элемента y множества P_i, не превосходящего t, можно указать элемент z в списке L_i, для которого

$$(1-\varepsilon/n)^i y \leqslant z \leqslant y.$$

В частности, при $i = n$ можно взять в качестве y оптимальное решение задачи о сумме подмножества и убедиться, что существует $z \in L_n$, для которого

$$(1-\varepsilon/n)^n y \leqslant z \leqslant y$$

тем самым ответ алгоритма (который заведомо не меньше этого z) будет не меньше $(1-\varepsilon/n)^n y$. Нам осталость только убедиться, что $(1-\varepsilon/n)^n \geqslant (1-\varepsilon)$.

В саом деле, продифферецировав $(1-\varepsilon/u)^u$ по u, убеждаемся, что это выражение возрастает как функция от u; остаются сравнить его значения при $u = 1$ и $u = n$.

[Это легко понять и без производных: в убывающей последовальности

$$1, (1-\delta), (1-\delta)^2, \ldots, (1-\delta)^n$$

второй член меньше первого (на δ), третий член меньше второго и т.п. Разности между соседними членами убывают, так как составляют фиксированную долю от уменьшаемого. Поэтому общее изменение при переходе от первого члена к последнему не превосходит $n\delta$ и $(1-\delta)^n \geqslant 1 - n\delta$. Остаётся положить $\delta = \varepsilon/n$.]

Теперь оценим время работы нашего алгоритма, для чего надо оценить длину списков L_i. После сокращения списка соседние его элементы отличаются как минимум в $1/(1-\varepsilon/n)$ раз. При этом *максимальный оставляемый в списке элемент не превосходит t, а второй элемент (первый равен нулю) не меньше 1. Отсюда видно, что число элементов в списке не больше*

$$\log_{1/(1-\varepsilon/n)} t + 2 = \frac{\ln t}{\ln(1-\varepsilon/n)} + 2 \leqslant$$

$$\leqslant \frac{n \log t}{\varepsilon} + 2.$$

(Мы использовали неравенство (2.10) на последнем шаге.) Эта оценка полиномиальна, так как $\lg t$ примерно равно числу битов в записи числа t. Осталось заметить, что время работы алгоритма Approx-Subset-Sum *ограничивается полиномом от n и от длины списков L_i.*

Упражнения

37.4-1 Докажите равенство (37.11).

37.4-2 Докажите неравенства (37.12) и (37.13).

37.4-3 Как следует изменить схему приближения, описанную в этом разделе, если мы рассматриваем все суммы, составленные из элементов заданного списка, не меньшие заданного числа t, и хотим найти минимальную из них (с заданной относительной ошибкой)?

Задачи

37-1 Упаковка в язики Имеется n объектов, причём i-ый объект имеет размер s_i, где $0 < s_i < 1$. Мы хотим упаковать их в минимальное число ящиков единичного размера (каждый из которых может вместить любое число объектов, суммарный размер которых не превосходит 1).

a. Доказите, что задача нахождения минимально необходимого числа ящиков является NP-трудной. (Указание. Сведите к ней задачу о сумме подмножества.)

b. Пусть $S! \sum_{i=1}^{n} s_i$ — суммарный размер всех предметов. Покажите, что необходимо по крайней мере $\lceil S \rceil$ ящиков.

Эвристика первого подходящего *предлагает рассматривать все предметы по очереди и помещать их в первый (в некотором порядке) ящик, в который они ещё помещаются.*

c. Покажите, что при таком подходе не более чем один ящик будет заполнен менее чем наполовину.

d. Докажите, что исло использованных ящиков при этом будет не более $\lceil 2S \rceil$.

e. Докажите, что эта эвристика даёт решение, не более чем в 2 раза худшее оптимального.

f. Придумать эффективную реализацию этого подхода, и оценить время работы получившегося алгоритма.

37-2 Размер максимальной клини и приближённые алгоритмы

Пусть $G = (V, E)$ — неориентированный граф. Определим его k-ую степень $G^{(k)}$ как неориентированный граф $(V^{(k)}, E^{(k)})$, вер-

шинами которого являются последовательности из k элементов V, причём две вершины $(v_1, v_2, \ldots, v_k)$ и $(w_1, w_2, \ldots, w_k)$ соединены ребром в том и только том случае, если при любом i вершины v_i и w_i соединены ребром в V или совпадают.

a. Докажите, что размер максимальной клики в графе $G^{(k)}$ является k-ой степенью размера максимальной клики в графе G.

b. Покажите, что если бы для задачи о клике существовал приближенный алгоритм с ошибкой не более в C раз (для некоторого фиксированного C), то для этой задачи существовала бы полностью полиномиальная схема приближения.

[Как установлено в работе S. Arora and S. Safra, Approximating clique is NP-complete, in Proc. of the 33rd IEEE Symp. on Foundation of Computer Science, *1992, такое возможно, лишь если P=NP. Этот результат (и родственные ему) является одним из самых замечательных достижений в теории сложности вычислений за последнее десятилетие.]*

37-3 Задача о покрытии множествами с весами

Рассмотрим обощение задачи в покрытии множествами, в котором каждому множеству из семейства $\mathcal{F}$ приписан некоторый вес. При этом весом покрытия считается суммарный вес входящих в него множеств, и мы хотим найти покрытие наименьшего веса.

Покажите, что имеется естественное обобщение рассмотренного нами жадного алгоритма, которое позволяет решить эту задачу с ошибкой не более чем в $H(d)$ раз, где d — максимальный размер множеств покрытия.

Замечания

Литература, посвященная приближённым алгоритмам, обширна; для начала можно прочесть книгу Гэри и Джонсона [79]. Другое превосходное введение в предмет — Пападимитриу и Стайглиц [154]. Лоулер, Ленстра, Ринной-Кан и Шмойс [133] подробно рассматривают задачу о коммивояжёре.

Согласно Пападимитриу и Стаглицу, алгоритм APPROX-VERTEX-COVER *был предложен Гаврилом (F. Gavril) и Яннакакисом (M. Yannakakis). Алгоритм* APPROX-TSP-TOUR *приводится в замечательной работе Розенкранца, Стирна и Льюиса [170]. Теорема 37.3 заимствована из работы Сахни и Гонсалеса [172]. Анализ жадной эвристики для задачи о покрытии множествами представляет собой упрощённый вариант более общих рассуждений из работы Хватала [42]; само утверждение имеется в работах Джонсона [113] и Ловаса [141]. Алгоритм* APPROX-SUBSET-SUM *и его анализ родственны алгоритмам для задачи о рюкзаке и сумме подмножества, рассмотренным в работе Ибарры и Кима [111].*

Индекс