Первый курс, весенний семестр 2017/18
Конспект лекций по алгоритмам

Собрано 20 июля 2018 г. в 12:18

Содержание

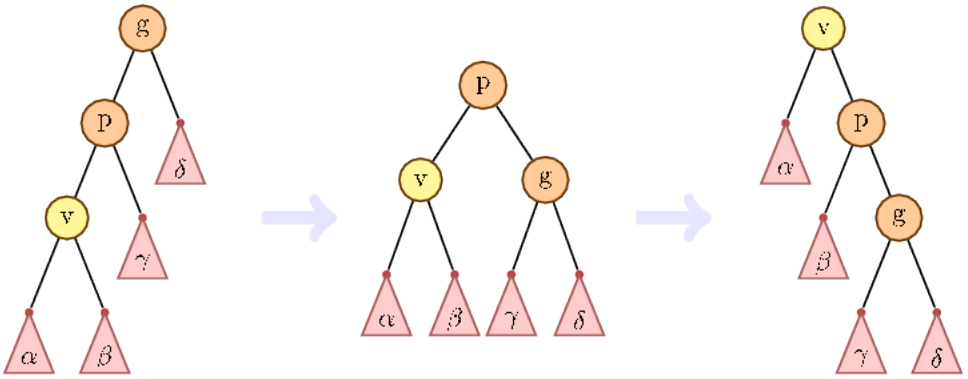
Лекция #1: Splay и корневая оптимизация

24 апреля

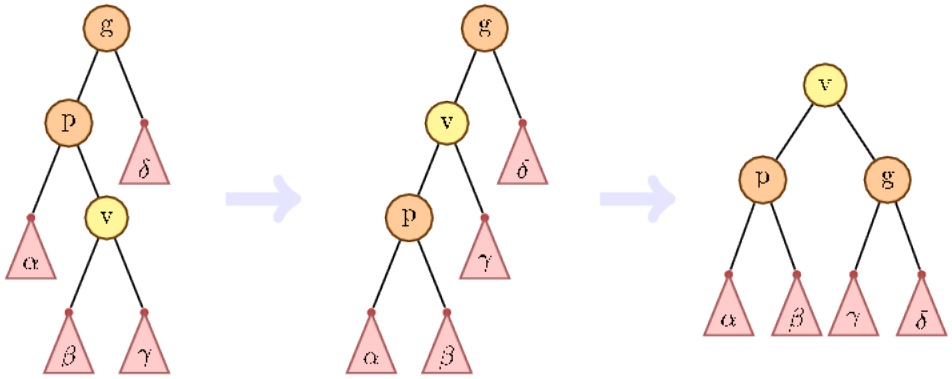
1.1. Splay tree

Splay-дерево — самобалансирующееся BST, не хранящее в вершине никакой дополнительной информации. В худшем случае глубина может быть линейна, но амортизированное время всех операций получится $\mathcal{O}(\log n)$. Возьмём обычное не сбалансированное дерево. При `add`/`del`. Модифицируем `add`: спустившись вниз до вершины v он самортизирует потраченное время вызовом `Splay(v)`, которая последовательными вращениями протолкнёт v до корня.

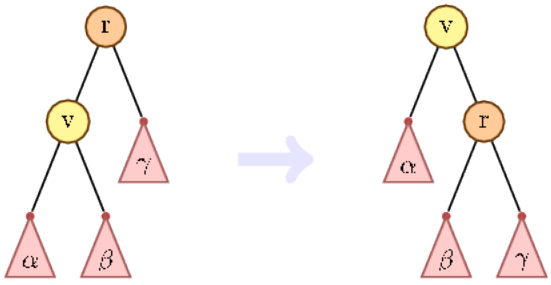
- **Zig-zig вращение**



- **Zig-zag вращение**



- **Если дедушки нет,** сделаем обычный single rotation (zig).



В частности из картинок видно, что все вращения выражаются через single rotation.

Любые другие операции со splay деревом делаются также, как и `add`: пусть v – самая глубокая вершина, до которой мы спустились $\Rightarrow$ вызовем `splay(v)`, который протолкнёт v в корень и самортизирует время работы. При этом всегда время `splay(v)` не меньше остальной полезной части $\Rightarrow$ осталось оценить время работы `splay`.

Lm 1.1.1. $x, y > 0, x + y = 1 \Rightarrow \log x + \log y \leqslant -2$

Доказательство. $\log x + \log y = \log x + \log(1 - x) = \log x(1 - x) \leqslant \log \frac{1}{2} \cdot 2 = -2$ ■

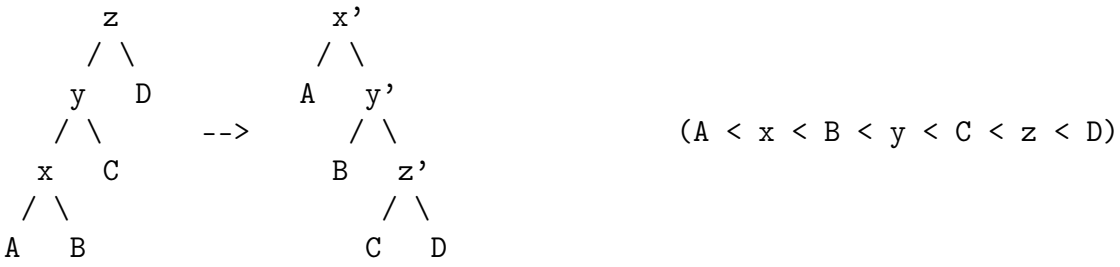
Lm 1.1.2. $x, y > 0, x + y = C \Rightarrow \log x + \log y \leqslant 2 \log C - 2$

Доказательство. $\log x + \log y = 2 \log C + \log \frac{x}{C} + \log \frac{y}{C} \leqslant 2 \log C - 2$ ■

Теперь введём потенциал. Ранг вершины $R_v = \log(size_v)$, где $size_v$ – размер поддерева. Потенциал $\varphi = \sum R_v$. Заметим сразу, что для пустого дерева $\varphi_0 = 0$ и $\forall$ момент времени $\varphi \geqslant 0$. Оценим амортизированное время операции `splay`, поднявшей v в u:

Теорема 1.1.3. $\forall v, u \; a_{v \to u} \leqslant 3(R_u - R_v) + 1 = 3 \log \frac{size_u}{size_v} + 1$

Доказательство. Полное доказательство доступно здесь. Мы разберём только случай zig-zig. Оставшиеся два аналогичны. +1 вылезет из случая zig (отсутствие деда). Итак, $a = t + \Delta\varphi = 2 + (R_{x'} + R_{y'} + R_{z'}) - (R_x + R_y + R_z) = 2 + R_{y'} + R_{z'} - R_x - R_y \leqslant 2 + R_{x'} + R_{z'} - 2R_x = F$.



Мы хотим показать $F \leqslant 3(R_{x'} - R_x) \Leftrightarrow R_{z'} \leqslant 2R_{x'} - R_x - 2 \Leftrightarrow R_{z'} + R_x \leqslant 2R_{x'} - 2$.
Теперь вспомним, что $R_{z'} = \log(C + D + 1), R_x = \log(A + B + 1) \overset{\text{лемма}}{\Rightarrow}$
$R_{z'} + R_x \leqslant 2 \log(A + B + C + D + 2) - 2 \leqslant 2R_{x'} - 2$. ■

Следствие 1.1.4. Среднее время одной операции в splay-дереве – $\mathcal{O}(\log n)$.

Доказательство. $\varphi_0 = 0, \varphi \geqslant 0 \Rightarrow \frac{1}{m} \sum t_i \leqslant a_i = \mathcal{O}(\log n)$. ■

Замечание 1.1.5. В теорему вместо $size_v$ можно подставить любой взвешенный размер:
$size_v = w_v + size_l + size_r$, где w_v – вес вершины.
Чтобы потенциал всегда был неотрицательный, потребуем $w_v \geqslant 1 \Rightarrow \log w_v \geqslant 0$.

Теперь о преимуществах splay-дерева. Элемент, к которому мы обращаемся чаще, оказывается ближе к корню, поэтому обращения к нему быстрее. Формально это можно записать так:

Теорема 1.1.6. Пусть к splay-дереву поступают только запросы `find(v)`, k_v – количество обращений к вершине v, $m = \sum k_v$. Тогда суммарное время всех `find` равно $\sum_v k_v(3 \log \frac{n+m}{k_v} + 1)$.

Доказательство. Внутри суммы k_v – количество запросов к v, $(3 \log \frac{n+m}{k_v} + 1)$ – время на один запрос. Эта оценка на время получается из 1.1.3 подстановкой веса вершины $w_v = \max(k_v, 1)$.
Тогда $size_{root} \leqslant n+m$ и время $splay(v) \leqslant 3 \log \frac{size_{root}}{size_v} + 1 \leqslant 3 \log \frac{size_{root}}{k_v} + 1$. ■

На практике мы также доказали теорему о времени работы бора (задача 11, есть разбор).

В той ж практике в 9-й задаче предлагается более быстрая `top-down` реализация splay-дерева.

1.2. SQRT decomposition

1.2.1. Корневая по массиву

Идея: разобьём массив на $\sqrt{n}$ частей (кусков) размера $k = \sqrt{n}$.

• Сумма на отрезке и изменение в точке

Решение: $\mathcal{O}(1)$ на изменение, $\mathcal{O}(\sqrt{n})$ на запрос суммы. $\forall i$ для i куска поддерживаем сумму `s[i]`.

```
1 void change(i, x):
2   s[i/k] += (x-a[i]), a[i]=x
3 int get(l, r): // [l, r)
4   int res = 0
5   while (l < r && l % k != 0) res += a[l++]; // левый хвост
6   while (l < r && r % k != 0) res += a[--r]; // правый хвост
7   return accumulate(s + l / k, s + r / k, res); // цельные куски
```

Запрос на отрезке разбивается на два хвоста длины $\sqrt{n}$ и не более $\sqrt{n}$ цельных кусков.

Решение за $\mathcal{O}(\sqrt{n})$ на изменение и $\mathcal{O}(1)$ на запрос суммы: будем поддерживать частичные суммы для каждого куска и для массива s. При изменении пересчитаем за $\mathcal{O}(\sqrt{n})$ частичные суммы внутри куска номер `i/k` и частичные суммы массива s. Суммы на хвостах и на отрезке массива s считаются за $\mathcal{O}(1)$.

• Минимум на отрезке и изменение в точке

Решение за $\mathcal{O}(\sqrt{n})$ на оба запроса – поддерживать минимум в каждом куске. В отличии от суммы, минимум мы сможем пересчитать только за $\mathcal{O}(\sqrt{n})$.

1.2.2. Корневая через split/merge

Дополним предыдущие две задачи операциями `insert(i,x)` и `erase(i)`.
Будем хранить `vector` или `list` кусков. Здесь i-й кусок – это отрезок $[l_i, r_i)$ нашего массива, мы храним его как `vector`. Сам массив мы не храним, только его разбиение на куски.
Когда приходит операция `insert`/`erase`, ищем, про какой она кусок за $\mathcal{O}(\sqrt{n})$. Теперь сделаем эту операцию в найденном куске за $\mathcal{O}(\sqrt{n})$. При этом кусок мог уменьшиться/увеличиться. Кусок размера меньше $\sqrt{n}$ смерджим с соседним. Кусок размера $2\sqrt{n}$ посплитим на два.
Поскольку мы удерживаем размер куска в $[\sqrt{n}, 2\sqrt{n})$, количество кусков всегда $\Theta(\sqrt{n})$.
Время старых операций не изменилось, время новых – $\mathcal{O}(\sqrt{n})$.

1.2.3. Корневая через split/rebuild

Храним то же, что и в прошлой задаче.
Операция `split(i)` – сделать так, чтобы i-й элемент был началом куска (если это не так, кусок, в котором лежит i, нужно разделить на два). В задачах про сумму и минимум `split` $= \mathcal{O}(\sqrt{n})$.
Ещё удобнее, если `split` возвращает номер куска, началом которого является i-й элемент.
Любой запрос на отрезке $[l, r)$ теперь будем начинать со `split(r), split(l)`.
И вообще хвостов нигде нет, всегда можно посплитить.
Тогда код любой функции теперь прекрасен своей лаконичностью:

```
vector<Part> p;
void insert(int i, int x) {
  i = split(i);
  a[n++] = x; // добавили x в конец исходного массива
  p.insert(i, Part(n - 1, n));
}
void erase(int i) {
  split(i + 1);
  p.erase(split(i));
}
int get_sum(int l, int r) { // [l,r)
  int sum = 0;
  for (r = split(r), l = split(l); l < r; l++)
    sum += p[l].sum;
  return sum;
}
```

Есть небольшая проблема – число кусков после каждого запроса вырастет на $\mathcal{O}(1)$.
Давайте, если число кусок $\geqslant 3\sqrt{n}$, просто вызовем `rebuild` – процедуру, которая выпишет все куски в один большой массив и от него с нуля построит структуру данных.
Время работы такой процедуры $\mathcal{O}(n)$, вызываем мы её не чаще чем раз в $\sqrt{n}$ запросов, поэтому среднее время работы `rebuild` – $\mathcal{O}(\sqrt{n})$ на запрос.

1.2.4. Применение

Задачи про минимум, сумму мы уже умели решать через BST, все операции за $\mathcal{O}(\log n)$.
Из нового мы пока научились только запросы сумма/изменение "перекашивать":
$\langle \mathcal{O}(\log n), \mathcal{O}(\log n) \rangle \longrightarrow \langle \mathcal{O}(\sqrt{n}), \mathcal{O}(1) \rangle$ и $\langle \mathcal{O}(1), \mathcal{O}(\sqrt{n}) \rangle$.
На самом деле спектр задач, решаемых корневой оптимизацией гораздо шире. Для примера приведём максимально ужасную задачу. Выполнять нужно следующие запросы:

1. `insert(i,x)`; `erase(i)`
2. `min(l,r)`
3. `reverse(l,r)`; `add_to_all(l,r,x)`
4. `sum(l,r,x,y)`; `kth_stat(l,r,k)`

Здесь `kth_stat(l,r,k)` – k-я статистика на отрезке. Бинпоиском по ответу такой запрос сводится к задаче вида `sum(l,r,x,y)` – число элементов на $[l, r)$ со значением от x до y. Чтобы отвечать на запрос `sum(l,r,x,y)` для каждого куска будем хранить его сортированную версию, тогда ответ на запрос – обработка двух хвостов за $\mathcal{O}(\sqrt{n})$ и $2\sqrt{n}$ бинпоисков. Итого $\sqrt{n} \log n$.

Чтобы отвечать на запросы `reverse(l,r)` и `add_to_all(l,r,x)` для каждого куска будем хранить две отложенных операции – `is_reversed` и `value_to_add`. Как пример, код `reverse(l,r)`:

```
void reverse(l, r) {
  r = split(r), l = split(l);
  reverse(p + l, p + r);
  for (; l < r; l++)
    p[l].is_reversed ^= 1;
}
```

Единственное место, где будет использоваться `is_reversed` – `split` куска.

Если мы хотим решать задачу через `split/merge`, чтобы выполнять операцию `reverse`, всё равно придётся добавить `split(i)`. Теперь можно делать `reverse(l,r)` ровно, как описано выше, после чего при наличии слишком маленьких кусков, сделаем им "`merge` с соседним".

1.2.5. Оптимальный выбор k

Не во всех задачах выгодно разбивать массив ровно на $\sqrt{n}$ частей по $\sqrt{n}$ элементов.
Обозначим число кусков k. В каждом куске $m = \frac{n}{k}$ элементов.
На примере последней задачи оптимизируем k.

• split/rebuild

Время `inner_split` куска: $\mathcal{O}(m \log m)$[1], так как нам нужно сортировать.
Время `split(i)`: $\mathcal{O}(k) +$ `inner_split` $= \mathcal{O}(k + m \log m)$.
Время `reverse` и `add_to_all`: $\mathcal{O}(k) +$ `split(i)` $= \mathcal{O}(k + m \log m)$.
Время `insert` и `erase`: $\mathcal{O}(k) + \mathcal{O}(m)$.
Время `sum`: хвосты и бинпоиски в каждом куске $= \mathcal{O}(k \log m) + \mathcal{O}(m)$.
Суммарное время всех запросов равно $\mathcal{O}((k + m) \log m)$.
В худшем случае нам будут давать все запросы по очереди $\Rightarrow$ эта асимптотика достигается.

Вспомним про `rebuild`! В этой задаче он работает за $\mathcal{O}(k(m \log m)) = \mathcal{O}(n \log n)$.
И вызывается он каждые k запросов (мы оцениваем только асимптотику, константу не пишем).

Итого: `T(split)` $+$ `T(insert)` $+$ `T(sum)` $+ \cdots +$ $\frac{1}{k}$`T(rebuild)` $= \Theta((k + m) \log m + \frac{1}{k} n \log n) \to \min$

При минимизации таких величин сразу замечаем, что "все log-и асимптотически равны".
$\frac{1}{k} n = m \Rightarrow$ минимизировать нужно $(\frac{n}{k} + k) \log n$. При минимизации $\frac{n}{k} + k$ мы не будем дифференцировать по k. Нас интересует только асимптотика, а $\Theta(f + g) = \Theta(\max(f, g))$.
Одна величина убывает, другая возрастает $\Rightarrow$ достаточно решить уравнение $\frac{n}{k} = k$.
Итого: $k = \sqrt{n}$, среднее время работы одного запроса $\mathcal{O}(\sqrt{n} \log n)$.

• split/merge

В этом случае всё то же, но нет `rebuild`.
Предположим, что мы умеем делать `inner_split` и `inner_merge` за $\mathcal{O}(m)$.
Тогда нам нужно минимизировать `T(split)` $+$ `T(insert)` $+$ `T(sum)` $+ \cdots = \Theta(m + k \log m)$
Заменили $\log m$ на $\log n$, сумму на максимум $\Rightarrow$ решаем $\frac{n}{k} = k \log n$. Итого $k = \sqrt{n / \log n}$.

1.2.6. Корневая по запросам, отложенные операции

Задача: даны числа, нужно отвечать на запросы `lower_bound`. Самое простое и быстрое решение – отсортировать числа, на сортированном массиве вызывать стандартный `lower_bound`.
Решение: отложенные операции, разобрано на 29-й странице осеннего конспекта.
Решение работает в online. Тем не менее, мы как будто обрабатываем запросы пачками по $\sqrt{n}$.

Другой пример на ту же тему – решение задачи dynamic connectivity в offline.
Задача: дан неорграф. Есть три типа запросов – добавить ребро в граф, удалить ребро, проверить связность двух вершин. Нужно в offline обработать m запросов.
Решение: обрабатывать запросы пачками по $\sqrt{m}$. Подробно описано в разборе практики (№7).

[1] при большом желании можно за $\mathcal{O}(m)$

Лекция #2: Mincost

2 октября 2017

2.1. Mincost k-flow в графе без отрицательных циклов

Сопоставим всем прямым рёбрам вес (стоимость) $w_e \in \mathbb{R}$.

Def 2.1.1. *Стоимость потока* $W(f) = \sum_e w_e f_e$. *Сумма по прямым рёбрам.*

Обратному к e рёбру $\overline{e}$ сопоставим $w_{\overline{e}} = -w_e$.

Если толкнуть поток сперва по прямому, затем по обратному к нему ребру, стоимость не изменится. Когда мы толкаем единицу потока по пути `path`, изменение потока и стоимости потока теперь выглядят так:

```
1 for (int e : path):
2   edges[e].f++
3   edges[e ^ 1].f--
4   W += edges[e].w;
```

Задача mincost k-flow: найти поток $f\colon |f| = k, W(f) \to \min$

При решении задачи мы будем говорить про веса путей, циклов, "отрицательные циклы", кратчайшие пути... Везде вес пути/цикла – сумма весов рёбер (w_e).

Решение #1. Пусть в графе нет отрицательных циклов, а также все $c_e \in \mathbb{Z}$.
Тогда по аналогии с алгоритмом Ф.Ф., который за $\mathcal{O}(k \cdot \texttt{dfs})$ искал поток размера k, мы можем за $\mathcal{O}(k \cdot \texttt{FordBellman})$ найти mincost поток размера k. Обозначим f_k оптимальный поток размера $k \Rightarrow f_0 \equiv 0, f_{k+1} = f_k + path$, где $path$ – кратчайший в G_{f_k}.

Lm 2.1.2. $\forall k, |f| = k \quad (W(f) = \min) \Leftrightarrow (\nexists$ отрицательного цикла в $G_f)$

Доказательство. Если отрицательный цикл есть, увеличим по нему поток, $|f|$ не изменится, $W(f)$ уменьшится. Пусть $\exists f^*\colon |f^*| = |f|, W(f^*) < W(f)$, рассмотрим поток $f^* - f$ в G_f.
Это циркуляция, мы можем декомпозировать её на циклы $c_1, c_2, \ldots, c_k$.
Поскольку $0 > W(f^* - f) = W(c_1) + \cdots + W(c_k)$, среди циклов c_i есть отрицательный. ■

Теорема 2.1.3. Алгоритм поиска mincost потока размера k корректен.

Доказательство. База: по условию нет отрицательных циклов $\Rightarrow f_0$ корректен.
Переход: обозначим f^*_{k+1} mincost поток размера $k{+}1$, смотрим на декомпозицию $\Delta f = f^*_{k+1} - f_k$.
$|\Delta f| = 1 \Rightarrow$ декомпозиция = путь p + набор циклов. Все циклы по 2.1.2 неотрицательны $\Rightarrow$ $W(f_k + p) \leqslant W(f^*_{k+1}) \Rightarrow$, добавив, кратчайший путь мы получим решение не хуже f^*_{k+1}. ■

Lm 2.1.4. Если толкнуть сразу $0 \leqslant x \leqslant \min_{e \in p}(c_e - f_e)$ потока по пути p,
то получим оптимальный поток размера $|f| + x$.

Доказательство. Обозначим f^* оптимальный поток размера $|f| + x$, посмотрим на декомпозицию $f^* - f$, заметим, что все пути в ней имеют вес $\geqslant W(p)$, а циклы вес $\geqslant 0$. ■

2.2. Потенциалы и Дейкстра

Для ускорения хотим Форда-Беллмана заменить на Дейкстру.
Для корректности Дейкстры нужна неотрицательность весов.
В прошлом семестре мы уже сталкивались с такой задачей, когда изучали алгоритм Джонсона.

• **Решение задачи mincost k-flow.**

Запустим один раз Форда-Беллмана из s, получим массив расстояний d_v, применим потенциалы d_v к весам рёбер:

$$e\colon a \to b \Rightarrow w_e \to w_e + d_a - d_b$$

Напомним, что из корректности d имеем $\forall e\ d_a + w_e \geqslant d_b \Rightarrow w_e^{'} \geqslant 0$.
Более того: для всех рёбер e кратчайших путей из s верно $d_a + w_e = d_b \Rightarrow w_e^{'} = 0$.

В G_f найдём Дейкстрой из s кратчайший путь p и расстояния d'_v.
Пустим по пути p поток, получим новый поток $f^{'} = f + p$.
В сети $G_f^{'}$ могли появиться новые рёбра (обратные к p). Они могут быть отрицательными.
Пересчитаем веса:

$$e\colon a \to b \Rightarrow w_e \to w_e + d'_a - d'_b$$

Поскольку d' – расстояния, посчитанные в G_f, все рёбра из G_f останутся неотрицательными.
p – кратчайший путь, все рёбра p станут нулевыми $\Rightarrow$ рёбра обратные p тоже будут нулевыми.

• **Псевдокод**

```
1 def applyPotentials(d):
2   for e in Edges:
3     e.w = e.w + d[e.a] - d[e.b]
4 d <-- FordBellman(s)
5 applyPotentials(d)
6 for i = 1..k:
7   d, path <-- Dijkstra(s)
8   for e in path: e.f += 1, e.rev.f -= 1
9   applyPotentials(d)
```

2.3. Графы с отрицательными циклами

Задача: найти mincost циркуляцию.

Алгоритм Клейна: пока в G_f есть отрицательный цикл, пустим по нему $\min_e(c_e - f_e)$ потока.

Пусть $\forall e\ c_e, w_e \in \mathbb{Z} \Rightarrow W(f)$ каждый раз уменьшается хотя бы на $1 \Rightarrow$ алгоритм конечен.

Задача: найти mincost k-flow циркуляцию в графе с отрицательными циклами.

Решение #1: найти за $|W(f)|$ итераций mincost циркуляцию, перейти от f_0 за k итераций к f_k.
Решение #2: найти любой поток $f\colon |f| = k$, в G_f найти mincost циркуляцию, сложить с f.

2.4. Mincost flow

Задача: найти $f\colon W(f) = \min$, размер f не важен.
Обозначим f_k – оптимальный поток размера k, p_k кратчайший путь в G_{f_k}.

Lm 2.4.1. $W(p_k)\nearrow$, как функция от k.

Доказательство. Аналогично доказательству леммы для Эдмондса-Карпа **??**.

От противного. Был поток f, мы увеличили его по кратчайшему пути p.
Расстояния в G_f обозначим d_0, в G_{f+p} – d_1.
Возьмём v: $d_1[v] < d_0[v]$, а из таких ближайшую к s в дереве кратчайших путей.
Рассмотрим кратчайший путь q в G_{f+p} из s в v: $s \rightsquigarrow \cdots \rightsquigarrow x \to v$.
$e = (v \to x), d_1[v] = d_1[x] + w_e, d_1[x] \geqslant d_0[x] \Rightarrow d_1[v] \geqslant d_0[x] + w_e \Rightarrow$ ребра $(x \to v)$ нет в $G_f \Rightarrow$
ребро $(v \to x) \in p \Rightarrow d_0[x] = d_0[v] + w_{\bar{e}} = d_0[v] - w_e \Rightarrow$
$d_1[v] = d_1[x] + w_e \geqslant d_0[x] + w_e = (d_0[v] - w_e) + w_e = d_0[v]$. Противоречие. ■

Следствие 2.4.2. $(W(f_k) = \min) \Leftrightarrow (W(p_{k-1}) \leqslant 0 \wedge W(p_k) \geqslant 0)$.

Осталось найти такое k бинпоиском или линейным поиском. На текущий момент мы умеем искать f_k или за $\mathcal{O}(k \cdot VE)$ с нуля, или за $\mathcal{O}(VE)$ из $f_{k-1} \Rightarrow$ линейный поиск будет быстрее.

2.5. Полиномиальные решения

Mincost flow мы можем бинпоиском свести к mincost k-flow.
Mincost k-flow мы можем поиском любого потока размера k свести к mincost циркуляции.
Осталось научиться за полином искать mincost циркуляцию.

• **Решение #1:** модифицируем алгоритм Клейна, будем толкать $\min_e(c_e - f_e)$ потока по циклу min среднего веса. Заметим, что (∃ отрицательный цикл) ⇔ (min средний вес < 0).

Решение работает за $\mathcal{O}(VE \log(nC))$ поисков цикла. Цикл ищется алгоритмом Карпа за $\mathcal{O}(VE)$. Доказано будет на практике.

• **Решение #2:** Capacity Scaling.

Начнём с графа $c'_e \equiv 0$, в нём mincost циркуляция тривиальна.
Будем понемногу наращивать c'_e и поддерживать mincost циркуляцию. В итоге хотим $c'_e \equiv c_e$.

```
1  for k = logU..0:
2    for e in Edges:
3      if c_e содержит бит 2^k:
4        c'_e += 2^k  // e: ребро из a_e в b_e
5        Найдём p - кратчайший путь a_e → b_e
6        if W(p) + w_e ⩾ 0:
7          нет отрицательных циклов ⇒ циркуляция f оптимальна
8        else:
9          пустим 2^k потока по циклу p + e (изменим f)
10         пересчитаем потенциалы, используя расстояния, найденные Дейкстрой
```

Время работы алгоритма $E \log U$ запусков Дейкстры $= E(E + V \log V) \log U$.

Lm 2.5.1. После 9-й строки циркуляция f снова минимальна.

Доказательство. f – минимальная циркуляция до 4-й строки, f' – после.
Как обычно, рассмотрим $f' - f$. Это тоже циркуляция. Декомпозируем её на единичные циклы.
Любой цикл проходит через e (иначе f не оптимальна). Через e проходит не более 2^k циклов.
Каждый из этих циклов имеет вес не меньше веса $p + e \Rightarrow W(f') \geqslant W(f + 2^k(p + e))$. ■

2.6. (*) Cost Scaling

Cost scaling (часть 1)
Cost scaling (часть 2)

Лекция #3: Суффиксный массив

16 октября 2017

Def 3.0.1. *Суффиксный массив s – отсортированный массив суффиксов s.*

Суффиксы сортируем в лексикографическом порядке. Каждый суффикс однозначно задается позицией начала в $s \Rightarrow$ на выходе мы хотим получить перестановку чисел от 0 до $n-1$.

• **Тривиальное решение:** `std::sort` отработает за $\mathcal{O}(n \log n)$ операций '<' $\Rightarrow$ за $\mathcal{O}(n^2 \log n)$.

3.1. Построение за $\mathcal{O}(n \log^2 n)$ хешами

Мы уже умеем сравнивать хешами строки на равенство, научимся сравнивать их на ">/<".
Бинпоиском за $\mathcal{O}(\log(min(|s|, |t|)))$ проверок на равенство найдём $x = lcp(s, t)$.
Теперь $less(s, t) = (s[x] < t[x])$. Кстати, в `C/C++` после строки всегда идёт символ с кодом 0.

Получили оператор меньше, работающий за $\mathcal{O}(\log n)$ и требующий $\mathcal{O}(n)$ предподсчёта.
Итого: суффмассив за $\mathcal{O}(n + (n \log n) \cdot \log n) = \mathcal{O}(n \log^2 n)$.

При написании сортировки нам нужно теперь минимизировать в первую очередь именно число сравнений $\Rightarrow$ с точки зрения `C++::STL` быстрее будет работать `stable_sort` (MergeSort внутри).

Замечание 3.1.1. Заодно научились за $\mathcal{O}(\log n)$ сравнивать на больше/меньше любые подстроки.

3.2. Применение суффиксного массива: поиск строки в тексте

Задача: дана строка t, приходят строки-запросы s_i: "является ли s_i подстрокой t".

Предподсчёт: построим суффиксный массив p строки t.
В суффиксом массиве сначала лежат все суффиксы $< s_i$, затем $\geqslant s_i \Rightarrow$ бинпоиском можно найти $\min k$: $t[p_k{:}] \geqslant s_i$. Осталось заметить, что (s_i – префикс $t[p_k{:}]$) $\Leftrightarrow$ (s_i – подстрока t).

Внутри бинпоиска можно сравнивать строки за линию, получим время $\mathcal{O}(|s_i| \log |t|)$ на запрос.
Можно за $\mathcal{O}(\log |t|)$ с помощью хешей, для этого нужно один раз предподсчитать хеши для t, а при ответе на запрос насчитать хеши s_i. Получили время $\mathcal{O}(|s_i| + \log |t| \cdot \log |s_i|)$ на запрос.

В разд. 3.6 мы улучшим время обработки запроса до $\mathcal{O}(|s_i| + \log |t|)$.

3.3. Построение за $\mathcal{O}(n^2)$ и $\mathcal{O}(n \log n)$ цифровой сортировкой

Заменим строку `s` на строку `s#`, где `#` – символ, лексикографически меньший всех в `s`.
Будем сортировать циклические сдвиги `s#`, порядок совпадёт с порядком суффиксом.
Длину `s#` обозначим n.

Решение за $\mathcal{O}(n^2)$: цифровая сортировка.
Сперва подсчётом по последнему символу, затем по предпоследнему и т.д.
Всего n фаз сортировок подсчётом. В предположении $|\Sigma| \leqslant n$ получаем время $\mathcal{O}(n^2)$.
Суффмассив, как и раньше задаётся перестановкой начал... теперь циклических сдвигов.

Решение за $\mathcal{O}(n \log n)$: цифровая сортировка с удвоением длины.
Пусть у нас уже отсортированы все подстроки длины k циклической строки `s#`.
Научимся за $\mathcal{O}(n)$ переходить к подстрокам длины $2k$.

Давайте требовать не только отсортированности но и знания "равны ли соседние в отсортированном порядке". Тогда линейным проходом можно для каждого i насчитать тип (цвет) циклического сдвига $c[i]$: $(0 \leqslant c[i] < n) \wedge (\texttt{s[i:i+k)} < \texttt{s[j:j+k)} \Leftrightarrow c[i] \leqslant c[j])$.

Любая подстрока длины $2k$ состоит из двух половин длины k $\Rightarrow$
переход $k \to 2k$ – цифровая сортировка пар $\langle c[i], c[i+k]\rangle$.

Прекратим удвоение k, когда $k \geqslant n$. Порядки подстрок длины k и n совпадут.

Замечание 3.3.1. В обоих решениях в случае $|\Sigma| > n$ нужно первым шагом отсортировать и перенумеровать символы строки. Это можно сделать за $\mathcal{O}(n \log n)$ или за $\mathcal{O}(n + |\Sigma|)$ подсчётом.

Реализация решения за $\mathcal{O}(n \log n)$.
`p[i]` – перестановка, задающая порядок подстрок длины `s[i:i+k)` циклической строки `s#`.
`c[i]` – тип подстроки `s[i:i+k)`.
За базу возьмём $k = 1$

```
bool sless( int i, int j ) { return s[i] < s[j]; }
sort(p, p + n, sless);
cc = 0; // текущий тип подстроки
for (i = 0; i < n; i++) // тот самый линейный проход, насчитываем типы строк длины 1
  cc += (i && s[p[i]] != s[p[i-1]]), c[p[i]] = cc;
```

Переход: (у нас уже отсортированы строки длины k) $\Rightarrow$ (уже отсортированы строки длины $2k$ по второй половине) $\Rightarrow$ (осталось сделать сортировку подсчётом по первой половине).

```
// pos - массив из n нулей
for (i = 0; i < n; i++)
  pos[c[i] + 1]++; // обойдёмся без лишнего массива cnt
for (i = 1; i < n; i++)
  pos[i] += pos[i - 1];
for (i = 0; i < n; i++) { // p[i] - позиция начала второй половины
  int j = (p[i] - k) mod n; // j - позиция начала первой половины
  p2[pos[c[j]]++] = j; // поставили подстроку s[j,j+2k) на правильное место в p2
}
cc = 0; // текущий тип подстроки
for (i = 0; i < n; i++) // линейным проходом насчитываем типы строк длины 2k
  cc += (i && pair_of_c(p2[i]) != pair_of_c(p2[i-1]])), c2[p2[i]] = cc;
c2.swap(c), p2.swap(p); // не забудем перейти к новой паре (p,c)
```

Здесь `pair_of_c(i)` – пара $\langle$`c[i], c[(i + k) mod n]`$\rangle$ (мы сортировали как раз эти пары!).

Замечание 3.3.2. При написании суффмассива в контесте рекомендуется, прочтя конспект, написать код самостоятельно, без подглядывания в конспект.

3.4. LCP за $\mathcal{O}(n)$: алгоритм Касаи

Алгоритм Касаи считает LCP соседних суффиксов в суффиксном массиве. Обозначения:

- $p[i]$ – элемент суффмассива,
- $p^{-1}[i]$ – позиция суффикса `s[i:]` в суффмассиве,
- $next_i = p[p^{-1}[i] + 1]$, $lcp_i = \mathrm{LCP}(i, next_i)$. Наша задача – насчитать массив lcp_i.

Утверждение 3.4.1. Если у i-го и j-го по порядку суффикса в суффмассиве совпадают первые k символов, то на всём отрезке $[i, j]$ суффмассива совпадают первые k символов.

Lm 3.4.2. Основная идея алгоритма Касаи: $lcp_i > 0 \Rightarrow lcp_{i+1} \geqslant lcp_i - 1$.

Доказательство. Отрежем у `s[i:]` и `s[next`$_i$`:]` по первому символу.
Получили суффиксы `s[i+1:]` и какой-нибудь `r`.
(`s[i:]` $\neq$ `s[next`$_i$`:]`) $\wedge$ (первый символ у них совпадал) $\Rightarrow$
(`r` в суффмассиве идёт после `s[i+1:]`) $\wedge$ (у них совпадает первых $lcp_i - 1$ символов) $\overset{3.4.1}{\Rightarrow}$
у `s[i+1:]` и `s[next`$_{i+1}$`]` совпадает хотя бы $lcp_i - 1$ символ $\Rightarrow lcp_{i+1} \geqslant lcp_i - 1$. ■

Собственно алгоритм заключается в переборе $i \searrow$ и подсчёте lcp_i начиная с $\max(0, lcp_{i+1} - 1)$.

Задача: уметь выдавать за $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$ LCP любых двух суффиксов строки s.

Решение: используем Касаи для соседних, а для подсчёта LCP любых других считаем RMQ. RMQ мы решили в прошлом семестре. Например, Фарах-Колтоном-Бендером за $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$.

3.5. Построение за $\mathcal{O}(n)$: алгоритм Каркайнена-Сандерса

На вход получаем строку s длины n, при этом $0 \leqslant s_i \leqslant \frac{3}{2}n$.
Выход – суффиксный массив. Сортируем именно суффиксы, а не циклические сдвиги.

Допишем к строке 3 нулевых символа. Теперь сделаем новый алфавит: $w_i = (s_i, s_{i+1}, s_{i+2})$.
Отсортируем w_i цифровой сортировкой за $\mathcal{O}(n)$, перенумеруем их от 0 до $n-1$.
Запишем все суффиксы строки s над новым алфавитом:

$t_0 = w_0 w_3 w_6 \dots$
$t_1 = w_1 w_4 w_7 \dots$
$t_2 = w_2 w_5 w_8 \dots$
$\dots$
$t_{n-1} = w_{n-1}$

Про суффиксы t_{3k+i}, где $i \in \{0, 1, 2\}$, будем говорить “суффикс i-типа”.

Запустимся рекурсивно от строки $t_0 t_1$. Длина $t_0 t_1$ не более $2\lceil \frac{n}{3} \rceil$.
Теперь мы умеем сравнивать между собой все суффиксы 0-типа и 1-типа.

Суффикс 2-типа = один символ + суффикс 0-типа $\Rightarrow$
их можно рассматривать как пары и отсортировать за $\mathcal{O}(n)$ цифровой сортировкой.

Осталось сделать merge двух суффиксных массивов.
Операция merge работает за линию, если есть “`operator <`”, работающий за $\mathcal{O}(1)$.
Нужно научиться сравнивать суффиксы 2-типа с остальными за $\mathcal{O}(1)$.

$\forall i, j\colon t_{3i+2} = s_{3i+2} t_{3i+3}$, $t_{3j} = s_{3j} t_{3j+1} \Rightarrow$ чтобы сравнить суффиксы 2-типа и 0-типа, достаточно уметь сравнивать суффиксы 0-типа и 1-типа. Умеем.

$\forall i, j\colon t_{3i+2} = s_{3i+2} t_{3i+3}$, $t_{3j+1} = s_{3j+1} t_{3j+2} \Rightarrow$ чтобы сравнить суффиксы 2-типа и 1-типа, достаточно уметь сравнивать суффиксы 0-типа и 2-типа. Только что научились.

• **Псевдокод.**

Пусть у нас уже есть `radixSort(a)`, возвращающий перестановку.

```
def getIndex(a): # новая нумерация, O(|a| + max_i a[i])
  p = radixSort(a)
  cc = 0
  ind = [0] * n
  for i in range(n):
    cc += (i > 0 and a[p[i]] != a[p[i-1]])
    ind[p[i]] = cc
  return ind

def sufArray(s): # 0 ⩽ s_i ⩽ 3/2 n
  n = len(s)
  if n < 3: return slowSlowSort(s)
  s += [0, 0, 0]
  w = getIndex( [(s[i], s[i+1], s[i+2]) for i in range(n)] )
  index01 = range(0, n, 3) + range(1, n, 3) # с шагом 3
  p01 = sufArray( [w[i] for i in index01] )
  pos = [0] * n
  for i in range(len(p01)): pos[index01[p01[i]]] = i # позиция 01-суффикса в p01
  index2 = range(2, n, 3)
  p2 = getIndex( [(w[i], pos[i+1]) for i in index2] )
  def less(i, j): # i mod 3 = 0/1, j mod 3 = 2
    if i mod 3 == 1: return (s[i],pos[i+1]) < (s[j],pos[j+1])
    else: return (s[i],s[i+1],pos[i+2]) < (s[j],s[j+1],pos[j+2])
  return merge(p01 ∘ index01, p2 ∘ index2, less) # ∘ - композиция: index01[p01[i]], ...
```

Для $n \geqslant 3$ рекурсивный вызов делается от строго меньшей строки:
$3 \to 1{+}1,\ 4 \to 2{+}1,\ 5 \to 2{+}2,\ \ldots$.

Неравенством $s_i \leqslant \frac{3}{2}n$ мы в явном виде в коде нигде не пользуемся.
Оно нужно, чтобы гарантировать, что `radixSort` работает за $\mathcal{O}(n)$.

3.6. Быстрый поиск строки в тексте

Представим себе простой бинпоиск за $\mathcal{O}(|s|\log(|text|))$. Будем стараться максимально переиспользовать информацию, полученную из уже сделанных сравнений.

Для краткости $\forall k$ обозначим k-й суффикс (`text[p`$_k$`:]`) как просто k.
Инвариант: бинпоиск в состоянии $[l, r]$ уже знает $lcp(s, l)$ и $lcp(s, r)$.
Сейчас мы хотим найти $lcp(s, m)$ и перейти к $[l, m]$ или $[m, r]$.
Заметим, $lcp(s, m) \geq max\{min\{lcp(s, l), lcp(l, m)\}, min\{lcp(s, r), lcp(r, m)\}\} = x$.
Мы умеем искать $lcp(l, m)$ и $lcp(r, m)$ за $\mathcal{O}(1) \Rightarrow$ `for (`$lcp(s, m) = x$`; можем;` $lcp(s, m)$`++)`.

Кстати, $lcp(l, m)$ и $lcp(r, m)$ не обязательно считать Фарах-Колтоном-Бендером, так как, аргументы lcp – не произвольный отрезок, а вершина дерева отрезков (состояние бинпоиска).
Предподсчитаем lcp для всех $\leqslant 2|text|$ вершин и по ходу бинпоиска будем спускаться по Д.О.

Теорема 3.6.1. Суммарное число увеличений на один $lcp(s, ?)$ не более $|x|$

Доказательство. Сейчас бинпоиск в состоянии l_i, m_i, r_i. Следующее состояние: l_{i+1}, r_{i+1}.
Предположим, $lcp(s, l_i) \geqslant lcp(s, r_i)$. Будем следить за величиной $z_i = \max\{lcp(s, l_i), lcp(s, r_i)\}$.
Пусть $lcp(s, m_i) < z_i \Rightarrow lcp(s, m) = x \wedge l_{i+1} = l_i \Rightarrow z_{i+1} = z_i$. Иначе $x = z_i \wedge z_{i+1} = lcp(s, m_i)$. ■

Лекция #4: Ахо-Корасик и Укконен

26 октября 2017

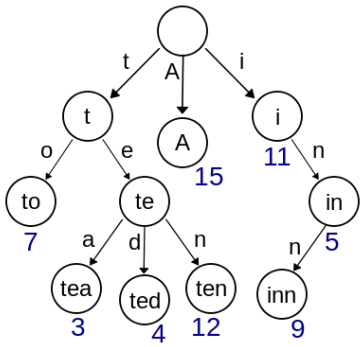
4.1. Бор

Бор – корневое дерево. Рёбра направлены от корня и подписаны буквами. Некоторые вершины бора подписаны, как конечные.

Базовое применение бора – хранение словаря `map<string, T>`.

Пример из wiki бора, содержащего словарь
$\{\text{A}:15, \text{to}:7, \text{tea}:3, \text{ted}:4, \text{ten}:12, \text{i}:11, \text{in}:5, \text{inn}:9\}$.

Для строки `s` операции `add(s)`, `delete(s)`, `getValue(s)` работают, как спуск вниз от корня.



Самый простой способ хранить бор: `vector<Vertex> t;`, где `struct Vertex { int id[|Σ|]; }`; Сейчас рёбра из вершины `t` хранятся в массиве `t.id[]`. Есть другие структуры данных:

Способ хранения	Время спуска по строке	Память на ребро
`array`	$\mathcal{O}(\|s\|)$	$\mathcal{O}(\|\Sigma\|)$
`list`	$\mathcal{O}(\|s\| \cdot \|\Sigma\|)$	$\mathcal{O}(1)$
`map (TreeMap)`	$\mathcal{O}(\|s\| \cdot \log \|\Sigma\|)$	$\mathcal{O}(1)$
`HashMap`	$\mathcal{O}(\|s\|)$ с большой const	$\mathcal{O}(1)$
`SplayMap`	$\mathcal{O}(\|s\| + \log S$	$\mathcal{O}(1)$

Иногда для краткости мы будем хранить бор массивом `int next[N][|Σ|];`. `next[v][c] == 0` $\Leftrightarrow$ ребра нет.

• Сортировка строк

Если мы храним рёбра в структуре, способной перебирать рёбра в лексикографическом порядке (не хеш-таблица, не список), можно легко отсортировать массив строк:
(1) добавить их все в бор, (2) обойти бор слева направо.
Для `SplayMap` и n и строк суммарной длины S, получаем время $\mathcal{O}(S + n \log S)$.
Для `TreeMap` получаем $\mathcal{O}(S \log |\Sigma|)$.

Замечание 4.1.1. Если бы мы научились сортировать строки над произвольным алфавитом за $\mathcal{O}(|S|)$, то для $\Sigma = \mathbb{Z}$, получилась бы сортировка целых чисел за $\mathcal{O}(|S|)$.

Часто размер алфавита считают $\mathcal{O}(1)$.
Например строчные латинские буквы – 26, или любимый для биологов $|\{A, C, G, T\}| = 4$.

4.2. Алгоритм Ахо-Корасика

Даны текст t и словарь $s_1, s_2, \ldots, s_m$, нужно научиться искать словарные слова в тексте.
Простейший алгоритм, отлично работающий для коротких слов, – сложить словарные слова в бор и от каждой позиции текста i попытаться пройти вперёд, откладывая суффикс t_i вниз по бору, и отмечая все концы слов, которые мы проходим. Время работы – $\mathcal{O}(|t| \cdot \max |s_i|)$.

Ту же асимптотику можно получить, сложив все хеши всех словарных слов в хеш-таблицу, и

проверив, есть ли в хеш-таблице какие-нибудь подстроки t длины не более $\max |s_i|$.

Давайте теперь соптимизируем первое решение также, как префикс-функция, позволяет простейший алгоритм поиска подстроки в строке улучшить до линейного времени. Обобщение префикс-функции на бор – суффиксные ссылки:

Def 4.2.1. $\forall$ *вершины бора* v:
$str[v]$ – *строка, написанная на пути от корня бора до* v.
$suf[v]$ – *вершина бора, соответствующая самому длинному суффиксу* $str[v]$ *в боре.*

$\forall$ позиции текста i насчитаем вершину бора v_i: $str[v_i]$ – суффикс $t[0{:}i)$, $|str[v_i]| \to \max$.
Пересчёт v_i**:**

```
v[0] = root, p = root
for (i = 0; i < |t|; i++)
  while (next[p][t[i]] == 0) // нет ребра
    p = suf[p]
  v[i + 1] = p = next[p][t[i]]
```

Чтобы цикл `while` всегда останавливался введём фиктивную вершину `f` и сделаем `suf[root] = f,` $\forall c$ `next[f][c] = root`.

Поиск словарных слов. Пометим все вершины бора, посещённые в процессе: `used[`v_i`] = 1`. В конце алгоритма поднимем пометки вверх по суффиксным ссылкам: `used[v]` $\Rightarrow$ `used[suf[v]]`. Для i-го словарного слова при добавлении мы запомнили вершину `end[i]`, тогда наличии этого слова в тексте лежит в `used[end[i]]`. Также можно насчитывать число вхождений.

Суффссылки. Чтобы всё это счастье работало осталось насчитать суффссылки.

Способ #1: полный автомат.

```
suf[v] = go[suf[parent[v]]][parent_char[v]];
go[v][c] = (next[v][c] ? next[v][c] : next[suf[v]][c]);
```

Естественно, чтобы от `parent[v]` и `suf[v]` всё было уже посчитано, поэтому нужно или перебирать вершины в порядке bfs от корня, или считать эту динамику рекурсивно-лениво.

Способ #2: пишем bfs от корня и пытаемся продолжить какой-нибудь суффикс отца.

```
q <-- root
while q --> v:
  z = suf[parent[v]]
  while next[z][parent_char[v]] == 0:
    z = suf[z]
  suf[v] = next[z][parent_char[v]]
```

Этот способ экономнее по памяти, если `next` – не массив, а, например, `map<int,int>`.

Теорема 4.2.2. Время построения линейно от длины суммарной строк, но не от размера бора.

Доказательство. Линейность от размера бора ломается на примере «бамбук длины n из букв a, из листа которого торчат рёбра по n разным символам». Линейность от суммарной длины строк следует из того, что если рассмотреть путь, соответствующий $\forall$ словарному слову s_i, то при вычислении суффссылок от вершин именно этого пути, указатель z в `while` всё время поднимался, а затем опускался не более чем на 1 $\Rightarrow$ сделал не более $2|s_i|$ шагов. ■

4.3. Суффиксное дерево, связь с массивом

Def 4.3.1. *Сжатый бор: разрешим на ребре писать не только букву, но и строку. При этом из каждой вершины по каждой букве выходит всё ещё не более одного ребра.*
Def 4.3.2. *Суффиксное дерево – сжатый бор построенный из суффиксов строки.*

Lm 4.3.3. Сжатое суффиксное дерево содержит не более $2n$ вершин.

Доказательство. Индукция: база один суффикс, 2 вершины, добавляем суффиксы по одному, каждый порождает максимум +1 развилку и +1 лист. ■

• Построение суффдерева из суффмассива+LCP

Пусть мы уже построили дерево из первых i суффиксов в порядке суффмассива. Храним путь от корня до конца i-го. Чтобы добавить $(i+1)$-й, поднимаемся до высоты $LCP(i, i+1)$ и делаем новую развилку, новый лист. Это несколько `pop`-ов и не более одного `push`-а. Итого $\mathcal{O}(n)$.

• Построение суффмассива+LCP из суффдерева

Считаем, что дерево построено от строки $s\$ \Rightarrow$ (листья = суффиксы).
Обходим дерево слева направо. Если в вершине используется неупорядоченный `map` для хранения рёбер, сперва отсортируем их. При обходе выписываем номера листьев-суффиксов.
$LCP(i, i+1)$ – максимально высокая вершина, из пройденных по пути из i в $i+1$.
Время работы $\mathcal{O}(n)$ или $\mathcal{O}(n \log |\Sigma|)$.

4.4. Суффиксное дерево, решение задач

• Число различных подстрок.

Это ровно суммарная длина всех рёбер. Так как любая подстрока есть префикс суффикса $\Rightarrow$ откладывается от корня дерева вниз до «середины» ребра.

• Поиск подстрок в тексте.

Строим суффдерево от текста. $\forall$ строку s можно за $\mathcal{O}(|s|)$ искать в тексте спуском по дереву.

• Общая подстрока k строк.

Построим дерево от $s_1\#_1 s_2\#_2 \ldots s_k\#_k$, найдём самую глубокую вершину, в поддереве которой содержатся суффиксы k различных типов. Время работы $\mathcal{O}(\sum |s_i|)$, оптимально по асимптотике. Константу времени работы можно улучшать за счёт уменьшения памяти – строить суффдерево не от конкатенации, а лишь от одной из строк.

4.5. Алгоритм Укконена

Обозначение: $ST(s)$ – суффиксное дерево строки s.
Алгоритм Укконена – онлайн алгоритм построения суффиксного дерево. Нам поступают по одной буквы c_i, мы хотим за амортизированное $\mathcal{O}(1)$ из $ST(s)$ получать $ST(sc_i)$.

За квадрат это делать просто: храним позиции концов всех суффиксов, каждый из них продлеваем вниз на c_i, если нужно, создаём при этом новые рёбра/вершины.

Ускорение #1: суффиксы, ставшие листьями, растут весьма однообразно – рассмотрим ребро $[l, r)$, за которое подвешен лист, тогда всегда происходит `r++`. Давайте сразу присвоим $[l, \infty)$.

Теперь опишем жизненный цикл любого суффикса:
рождается в корне, ползёт вниз по дереву, разветвляется, становится саморастущим листом.

Нам интересно обработать только момент разветвления.

Lm 4.5.1. ⌝ Суффикс длины k не разветвился $\Rightarrow$ все более короткие тоже не разветвились.

Доказательство. Суффикс длины k не разветвился $\Rightarrow$ он встречался в s как подстрока. Все более короткие являются его суффиксами $\Rightarrow$ тоже встречаются в s $\Rightarrow$ не разветвятся. ■

Ускорение #2: давайте хранить только позицию самого длинного неразветвившегося суффикса. Пока он спускается по дереву, ничего не нужно делать. Как только он разветвится, нужно научиться быстро переходить к следующему по длине (отрезать первую букву).

Ускорение #3: отрезать первую букву = перейти по суффссылке, давайте от всех вершин поддерживать суффссылки. Если мы были в вершине, когда не смогли пойти вниз, теперь всё просто, перейдём по её суффссылке. Если же мы стояли посередине ребра и создали новую вершину v, от неё следует посчитать суффссылку. Для этого возьмём суффссылку её отца $p[v]$ и из $suf[p[v]]$ спустимся вниз на строку, соединяющую $p[v]$ и v.

```
1  void build( char *s ) {
2    int N = strlen(s), VN = 2 * Ns;
3    int vn = 2, v = 1, pos; // идём по ребру из p[v] в v, сейчас стоим в pos
4    int suf[VN], l[VN], r[VN], p[VN]; // «ребро p[v] → v» = s[l[v]:r[v])
5    map<char,int> t[VN]; // собственно рёбра нашего бора
6    for (int i = 0; i < |Σ|; i++) t[0][i] = 1; // 0 = фиктивная, 1 = корень
7    l[1] = -1, r[1] = 0, suf[1] = 0;
8    for (int n = 0; n < N; n++) {
9      char c = s[n];
10     auto new_leaf = [&]( int v ) {
11       p[vn] = v, l[vn] = n, r[vn] = ∞, t[v][c] = vn++;
12     };
13     go:;
14     if (r[v] <= pos) { // дошли до вершины, конца ребра
15       if (!t[v].count(c)) { // по символу c нет ребра вперёд, создаём
16         new_leaf(v), v = suf[v], pos = r[v];
17         goto go;
18       }
19       v = t[v][c], pos = l[v] + 1; // начинаем идти по новому ребру
20     } else if (c == s[pos]) {
21       pos++; // спускаемся по ребру
22     } else {
23       int x = vn++; // создаём развилку
24       l[x] = l[v], r[x] = pos, l[v] = pos;
25       p[x] = p[v], p[v] = x;
26       t[p[x]][s[l[x]]] = x, t[x][s[pos]] = v;
27       new_leaf(x);
28       v = suf[p[x]], pos = l[x]; // вычисляем позицию следующего суффикса
29       while (pos < r[x])
30         v = t[v][s[pos]], pos += r[v] - l[v];
31       suf[x] = (pos == r[x] ? v : vn);
32       pos = r[v] - (pos - r[x]);
33       goto go;
34     }
35   }
36 }
```

Теорема 4.5.2. Суммарное время работы n первых шагов равно $\mathcal{O}(n)$.

Доказательство. Понаблюдаем за величиной φ "число вершин на пути от корня до нас". Пока мы идём вниз, φ растёт, когда переходим по суффссылке, φ уменьшается максимум на 1 $\Rightarrow$ суммарное число шагов вниз не больше n. ■

4.6. LZSS

Решим ещё одну задачу – сжатие текста алгоритмом LZSS.
В отличии от использования массива, дерево даёт чисто линейную асимптотику и простейшую реализацию – насчитаем для каждой вершины $l[v]$ = самый левый суффикс в поддереве и при попытки найти $j < i$: $LCP(j, i) = \max$ будем спускаться из корня, пока $l[v] < i$.

Лекция #5: Быстрое преобразование Фурье

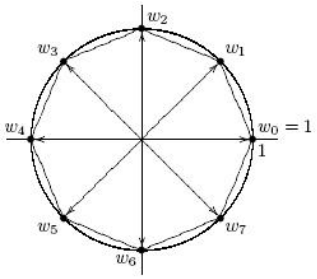
4 декабря 2017

Перед тем, как начать говорить "Фурье" то, "Фурье" сё, нужно сразу заметить:
Есть непрерывное преобразование Фурье. С ним вы должны столкнуться на теорвере.
Есть тригонометрический ряд Фурье. И есть общий ряд Фурье в гильбертовом пространстве, который появляется в начале курса функционального анализа.
Мы же с вами будем заниматься исключительно дискретным преобразованием Фурье.
Коротко DFT (Discrete Fourier transform). FFT – по сути то же, первая буква означает "fast".

Задача: даны два многочлена A, B суммарной длины $\leqslant n$, перемножить их за $\mathcal{O}(n \log n)$.

Длина многочлена – $\gamma(A) = (\deg A) + 1$. Вводим её, чтобы везде не писать "-1".
Если даны n точек (x_i, y_i), все x_i различны, $\exists!$ интерполяционный многочлен длины n, построенный по этим точкам (из алгебры). Ещё заметим: $\gamma(AB) = \gamma(A) + \gamma(B) - 1$. Наш план:

1. Подобрать удачные k и точки $w_0, w_1, \ldots, w_{k-1}$: $k \geqslant \gamma(A) + \gamma(B) - 1 = n$.
2. Посчитать значения A и B в w_i.
3. $AB(x_i) = A(x_i)B(x_i)$. Эта часть самая простая, работает за $\mathcal{O}(n)$.
4. Интерполировать AB длины k по полученным парам $\langle w_i, AB(w_i)\rangle$.



Вспомним комплексные числа:

$e^{2\pi i\alpha} e^{2\pi i\beta} = e^{2\pi i(\alpha+\beta)}$ $e^{2\pi i\varphi} = (\cos\varphi, \sin\varphi)$, $\overline{(a,b)} = (a,-b) \Rightarrow \overline{e^{2\pi i\varphi}} = e^{2\pi i(-\varphi)}$
Извлечение корня k-й степени: $\sqrt[k]{z} = \sqrt[k]{e^{2\pi i\varphi}} = e^{2\pi i\varphi/k}$
Если взять все корни из 1 степени 2^t, возвести в квадрат,
получатся ровно все корни степени 2^{t-1}. Корни из 1 степени k: $e^{2\pi ij/k}$.

5.1. Прелюдия к FFT

Возьмём $\min N = 2^t \geqslant n$ и $w_j = e^{2\pi ij/N}$. Тут мы предполагаем, что $A, B \in \mathbb{C}[x]$ или $A, B \in \mathbb{R}[x]$.
Пусть есть многочлены $A(x) = \sum a_i x^i$ и $B(x) = \sum b_i x^i$. Ищем $C(x) = A(x)B(x)$.
Обозначим их значения в точках $w_0, w_1, \ldots, w_{k-1}$: $A(w_i) = fa_i, B(w_i) = fb_i, C(w_i) = fc_i$.
Схема быстрого умножения многочленов:

$$a_i, b_i \xrightarrow{\mathcal{O}(n\log n)} fa_i, fb_i \xrightarrow{\mathcal{O}(n)} fc_i = fa_i fb_i \xrightarrow{\mathcal{O}(n\log n)} c_i$$

5.2. Собственно идея FFT

$A(x) = \sum a_i x^i = (a_0 + x^2 a_2 + x^4 a_4 + \ldots) + x(a_1 x + a_3 x^3 + a_5 x^5 + \ldots) = P(x^2) + xQ(x^2)$
Т.е. обозначили все чётные коэффициенты A многочленом P, а нечётные соответственно Q.

$\gamma(A) = n$, все $w_j^2 = w_{n/2+j}^2 \Rightarrow$ многочлены P и Q нужно считать не в n, а в $\frac{n}{2}$ точках.

```
1 def FFT(a):
2   n = len(a)
3   if n == 1: return a[0] # посчитать значение A(x) = a[0] в точке 1
4   a ---> p, q # разбили коэффициенты на чётные и нечётные
5   p, q = FFT(p), FFT(q)
6   w = exp(2pi*i/n) # корень из единицы n-й степени
7   for i=0..n-1: a[i] = p[i%(n/2)] + w^i*q[i%(n/2)]
8   return a
```

Время работы $T(n) = 2T(n/2) + \mathcal{O}(n) = \mathcal{O}(n \log n)$.

5.3. Крутая реализация FFT

Чтобы преобразование работало быстро, нужно заранее предподсчитать все $w_j = e^{2\pi ij/N}$.
Заметим, что p и q можно хранить прямо в массиве a.
Тогда получается, что на прямом ходу рекурсии мы просто переставляем местами элементы a, и только на обратном делаем какие-то полезные действия.
Число a_i перейдёт на позицию $a_{rev(i)}$, где $rev(i)$ – перевёрнутая битовая запись i.
Кстати, $rev(i)$ мы уже умеем считать динамикой для всех i.
При реализации на C++ можно использовать комплексные числа из STL: `complex<double>`.

```
const int K = 20, N = 1 << K;
complex<double> root[N];
int rev[N];
void init() {
  for (int j = 0; j < N; j++) {
    root[j] = exp(2π*i*j/N); // cos(2πj/N), sin(2πj/N)
    rev[j] = (rev[j >> 1] >> 1) + ((j & 1) << (K - 1));
  }
}
```

Теперь, корни из единицы степени k хранятся в `root[j*N/k]`, $j \in [0, k)$. Две проблемы:
1. Доступ к памяти при этом не последовательный, проблемы с кешом.
2. Мы $2N$ раз вычисляли тригонометрические функции.
$\Rightarrow$ можно лучше, вычисления корней #2:

```
for (int k = 1; k < N; k *= 2) {
  num tmp = exp(π/k);
  root[k] = {1, 0}; // в root[k..2k) хранятся первые k корней степени 2k
  for (int i = 1; i < k; i++)
    root[k + i] = (i & 1) ? root[(k + i) >> 1] * tmp : root[(k + i) >> 1];
}
```

Теперь код собственно преобразования Фурье может выглядеть так:

```
void FFT(a, fa) { // a --> fa
  for (int i = 0; i < N; i++)
    fa[rev[i]] = a[i]; // можно иначе, но давайте считать массив «a» readonly
  for (int k = 1; k < N; k *= 2) // уже посчитаны FFT от кусков длины k, база: k=1
    for (int i = 0; i < N; i += 2 * k) // [i..i+k) [i+k..i+2k) --> [i..i+2k)
      for (int j = 0; j < k; j++) { // оптимально написанный стандартный цикл FFT
        num tmp = root[k + j] * fa[i + j + k]; // вторая версия root[]
        fa[i + j + k] = fa[i + j] - tmp; // exp(2πi(j+n/2)/n) = -exp(2πij/n)
        fa[i + j] = fa[i + j] + tmp;
      }
}
```

5.4. Обратное преобразование

Теперь имея при $w = e^{2\pi i/n}$:
$fa_0 = a_0 + a_1 + a_2 + a_3 + \dots$
$fa_1 = a_0 + a_1 w + a_2 w^2 + a_3 w^3 + \dots$
$fa_2 = a_0 + a_1 w^2 + a_2 w^4 + a_3 w^3 + \dots$
$\dots$
Нам нужно научиться восстанавливать коэффициенты $a_0, a_1, a_2, \dots$, имея только fa_i.
Заметим, что $\forall j \neq 0 \sum\limits_{k=0}^{n-1} w^{jk} = 0$ (геометрическая прогрессия). А при $j = 0$ получаем $\sum\limits_{k=0}^{n-1} w^{jk} = n$.

Поэтому $fa_0 + fa_1 + fa_2 + \dots = a_0 n + a_1 \sum_k w^k + a_2 \sum_k w^{2k} + \dots = a_0 n$
Аналогично $fa_0 + fa_1 w^{-1} + fa_2 w^{-2} + \dots = \sum_k a_0 w^{-k} + a_1 n + a_2 \sum_k w^k + \dots = a_1 n$
И в общем случае $\sum_k fa_k w^{-jk} = a_j n$.
Заметим, что это ровно значение многочлена с коэффициентами fa_k в точке w^{-j}.
Осталось заметить, что множества чисел $\{w_j \mid j = 0..n{-}1\}$ и $\{w_{-j} \mid j = 0..n{-}1\}$ совпадают $\Rightarrow$

```
1 void FFT_inverse(fa, a) { // fa --> a
2   FFT(fa, a)
3   reverse(a + 1, a + N) // w^j <--> w^{-j}
4   for (int i = 0; i < N; i++) a[i] /= N;
5 }
```

5.5. Два в одном

Часто коэффициенты многочленов – вещественные числа.
Если у нас есть многочлены $A(x), B(x) \in \mathbb{R}[x]$, возьмём числа $c_j = a_j + ib_j$ и посчитаем $fc = FFT(c)$. Тогда по fc за $\mathcal{O}(n)$ можно легко восстановить fa и fb.

Для этого вспомним про сопряжения комплексных чисел:
$\overline{x + iy} = x - iy, \overline{a \cdot b} = \overline{a} \cdot \overline{b}, w^{n-j} = w^{-j} = \overline{w^j} \Rightarrow \overline{fc_{n-j}} = \overline{C(w^{n-j})} = \overline{C}(w^j) \Rightarrow$
$fc_j + \overline{fc_{n-j}} = 2 \cdot A(w^j) = 2 \cdot fa_j$. Аналогично $fc_j - \overline{fc_{n-j}} = 2i \cdot B(w^j) = 2i \cdot fb_j$.

Теперь, например, для умножения двух $\mathbb{R}[x]$ можно использовать не 3 вызова FFT, а 2.

5.6. Умножение чисел, оценка погрешности

Общая схема умножения чисел:
цифра – коэффициент многочлена ($x = 10$); умножим многочлены; сделаем переносы.

Число длины n в системе счисления 10 можно за $\mathcal{O}(n)$ перевести в систему счисления 10^k. Тогда многочлены будут длины n/k, умножение многочленов работать за $\frac{n}{k} \log \frac{n}{k}$ (убывает от k).
Возникает вопрос, какое максимальное k можно использовать?
Коэффициенты многочлена-произведения будут целыми числами до $(10^k)^2 \frac{n}{k}$.
Чтобы в типе `double` целое число хранилось с погрешностью меньше 0.5 (тогда его можно правильно округлить к целому), оно должно быть не более 10^{15}.
Получаем при $n \leqslant 10^6$, что $(10^k)^2 10^6 / k \leqslant 10^{15} \Rightarrow k \leqslant 4$.
Аналогично для типа `long double` имеем $(10^k)^2 10^6 / k \leqslant 10^{18} \Rightarrow k \leqslant 6$.
Это оценка сверху, предполагающая, что само FFT погрешность не накапливает... на самом деле эта оценка очень близка к точной.

Лекция #6: Длинная арифметика

11 декабря 2017

Мы займёмся целыми беззнаковыми числами. Целые со знаком – ещё отдельно хранить знак. Вещественные – то же, но ещё хранить экспоненту: $12.345 = 12345e{-}3$, мы храним 12345 и -3.

Удобно хранить число в "массиве цифр", младшие цифры в младших ячейках.
Во примерах ниже мы выбираем систему счисления $\texttt{BASE} = 10^k$, $k \to \max$: нет переполнений.
Пусть есть длинное число a. При оценки времени работы будем использовать обозначения:
$|a| = n$ – битовая длина числа и $\frac{n}{k}$ – длина числа, записанного в системе 10^k. Помните, $\max k \approx 9$.

Если мы ленивы и уверены, что в процессе вычислений не появятся числа длиннее N,
наш выбор – `int[N];`, иначе обычно используют `vector<int>` и следят за длиной числа.

Примеры простейших операций:

```
1  const int N = 100, BASE = 1e9, BASE_LEN = 9;
2  void add( int *a, int *b ) { // сложение за O(n/k)
3    for (int i = 0; i + 1 < N; i++) // +1, чтобы точно не было range check error
4      if ((a[i] += b[i]) >= BASE)
5        a[i] -= BASE, a[i + 1]++;
6  }
7  int divide( int *a, int k ) { // деление на короткое за O(n/k), делим со старших разрядов
8    long long carry = 0; // перенос с более старшего разряда, он же остаток
9    for (int i = N - 1; i >= 0; i--) {
10     carry = carry * BASE + a[i]; // максимальное значение carry < BASE^2
11     a[i] = carry / k, carry %= k;
12   }
13   return carry; // как раз остаток
14 }
15 int mul_slow( int *a, int *b, int *c ) { // умножение за (n/k)^2
16   fill(c, c + N, 0);
17   for (int i = 0; i < N; i++)
18     for (int j = 0; i + j < N; j++)
19       c[i + j] += a[i] * b[j]; // здесь почти наверняка произойдёт переполнение
20   for (int i = 0; i + 1 < N; i++) // сначала умножаем, затем делаем переносы
21     c[i + 1] += c[i] / BASE, c[i] %= BASE;
22 }
23 void out( int *a ) { // вывод числа за O(n/k)
24   int i = 0;
25   while (i && !a[i]) i--;
26   printf("%d", a[i--]);
27   while (i >= 0) printf("%0*d", BASE_LEN, a[i--]); // воспользовались таки BASE_LEN!
28 }
```

Чтобы в строке 19 не было переполнения, нужно выбирать `BASE` так, что $\texttt{BASE}^2$ `N` помещалось в тип данных. Например, хорошо сочетаются $\texttt{BASE} = 10^8, \texttt{N} = 10^3$, тип – `unsigned long long`.

6.1. Бинарная арифметика

Пусть у нас реализованы простейшие процедуры: "+, -, *2, /2, %2, >, ⩾, isZero".
Давайте выразим через них "*, \, gcd". Обозначим $|a| = n, |b| = m$.

Умножение будет полностью изоморфно бинарному возведению в степень.

```
num mul(num a, num b) {
  if (isZero(b)) return 1; // если храним число, как vector, то isZero за O(1)
  num res = mul(mul(mul2(a), div2(b));
  if (mod2(b) == 1) add(res, a); // функция mod2 всегда за O(1)
  return res;
}
```

Глубина рекурсии равна m. В процессе появляются числа не более $(n+m)$ бит длины $\Rightarrow$ каждая операция выполняется за $\mathcal{O}(\frac{n+m}{k}) \Rightarrow$ суммарное время работы $\mathcal{O}((n+m)\frac{m}{k})$.
Если большее умножать на меньшее, то $\mathcal{O}(\max(n,m)\min(n,m)/k)$.

Деление в чём-то похоже... деля a на b, мы будем пытаться вычесть из a числа $b, 2b, 4b, \ldots$.

```
pair<num, num> div(num a, num b) { // найдём для удобства и частное, и остаток
  num c = 1, res = 0;
  while (b < a) // (n−m) раз
    mul2(b), mul2(c);
  while (!isZero(c)) { // Этот цикл сделает ≈ n−m итераций
    if (a >= b) // O(n), так как длины a и b убывают от n до 1
      sub(a, b), add(res, c); O(n)
    div2(b), div2(c); O(n)
  }
  return {res, a};
}
```

Глубина рекурсии равна $n-m$. Все операции за $\mathcal{O}(\frac{n}{k}) \Rightarrow$ суммарное время $\mathcal{O}((n-m)\frac{n}{k})$.

Наибольший общий делитель сделаем самым простым Евклидом "с вычитанием".
Добавим только одну оптимизацию: если числа чётные, надо сразу их делить на два...

```
num gcd(num a, num b) {
  int pow2 = 0;
  while (mod2(a) == 0 && mod2(b) == 0)
    div2(a), div2(b), pow2++;
  while (!isZero(b)) {
    while (mod2(a) == 0) div2(a);
    while (mod2(b) == 0) div2(b);
    if (a < b) swap(a, b);
    a = sub(a, b); // одно из чисел станет чётным
  }
  while (pow2--) mul2(a);
  return a;
}
```

Шагов главного цикла не больше $n+m$. Все операции выполняются за $\max(n,m)/k$.
Отсюда суммарное время работы: $\mathcal{O}(\max(n,m)^2/k)$.

6.2. Деление многочленов за $\mathcal{O}(n \log^2 n)$

Коэффициенты многочлена $A(x)$: $A[0]$ – младший, $A[\deg A]$ – старший. $\gamma(A) = \deg A - 1$.

Задача: даны $A(x), B(x) \in \mathbb{R}[x]$, найти $Q(x), R(x)$: $\deg R < \deg B \wedge A(x) = B(x)Q(x) + R(x)$.

Сперва простейшее решение за $\mathcal{O}(\deg A \cdot \deg B)$, призванное побороть страх перед делением:

```
pair<F*,F*> divide( int n, F *a, int m, F *b ) { // deg A = n, deg B = m, F - поле
  F q[n-m+1];
  for (int i = n - m; i >= 0; i--) { // выводим коэффициенты частного в порядке убывания
    q[i] = a[i + m] / b[m]; // m - степень ⇒ b[m] ≠ 0.
    for (int j = 0; j <= m; j--) // конечно, вычитать имеет смысл, только если q[i] ≠ 0
      a[i + j] -= b[j] * q[i]; // можно соптимизить, перебирать только ненулевые b[j]
  }
  return {q, a}; // в a как раз остался остаток
}
```

Теперь перейдём к решению за $\mathcal{O}(n \log^2 n)$.
Зная Q, мы легко найдём R, как $A(x) - B(x)Q(x)$ за $\mathcal{O}(n \log n)$. Сосредоточимся на поиске Q.
Пусть $\deg A = \deg B = n$, тогда $Q(x) = \frac{a_n}{b_n}$. То есть, $Q(x)$ можно найти за $\mathcal{O}(1)$.
Из этого мы делаем вывод, что Q зависит не обязательно от всех коэффициентов A и B.

Lm 6.2.1. $\deg A = m, \deg B = n \Rightarrow \deg Q = m - n$, и Q зависит только от $m-n+1$ старших коэффициентов A и $m-n+1$ коэффициентов B.

Доказательство. Рассмотрим деление в столбик, шаг которого: A `-=` $\alpha x^i B$. $\alpha = \frac{A[i+\deg B]}{B[\deg B]}$.
Поскольку $i + \deg B \geqslant \deg B = n$, младшие n коэффициентов A не будут использованы. ■

Теперь будем решать задачу:
Даны $A, B \in \mathbb{R}[x]$: $\gamma(A) = \gamma(B) = n$, найти $C \in \mathbb{R}[x]$: $\gamma(C) = n$,
что у A и BC совпадает n старших коэффициентов.

```
int* Div( int n, int *A, int *B ) // n - степень двойки (для удобства)
  C = Div(n/2, A + n/2, B + n/2) // нашли старших n/2 коэффициентов ответа
  A' = Subtract(n, A, n + n/2 - 1, Multiply(C, B))
  D = Div(n/2, A', B + n/2) // сейчас A' состоит из n/2 не нулей и n/2 нулей
  return concatenate(D, C) // склеили массивы коэффициентов; вернули массив длины ровно n
```

Здесь `Subtract` – хитрая функция. Она знает длины многочленов, которые ей передали, и сдвигает вычитаемый многочлен так, чтобы старшие коэффициенты совместились.

Время работы: $T(n) = 2T(n/2) + \mathcal{O}(n \log n) = \mathcal{O}(n \log^2 n)$. Здесь $\mathcal{O}(n \log n)$ – время умножения.

6.3. Деление чисел

Оптимально использовать метод Ньютона, внутри которого все умножения – FFT.
Тогда мы получим асимптотику $\mathcal{O}(n \log n)$. Об этом можно будет узнать на третьем курсе.
Сегодня лучшими результатами будут $\mathcal{O}((n/k)^2)$ и $\mathcal{O}(n \log^2 n)$.

Простейшие методы (оценка времени деление числа битовой длины $2n$ на число длины n).

1. Бинпоиск по ответу: n^3/k^2 при простейшем умножении, $n^2 \log n$ при Фурье внутри.
2. Двоичное деление: n^2/k времени.
3. Деление в столбик: n^2/k^2 времени. На нём остановимся подробнее.

6.4. Деление чисел за $\mathcal{O}((n/k)^2)$

Делить будем в столбик. У нас уже было деление многочленов за квадрат. Если мы научимся вычислять за $\mathcal{O}(n/k)$ старшую цифру частного, мы сможем воспользоваться им без изменений. Пусть даны числа $a, b, |a| = n, |b| = m$.

Lm 6.4.1. Старшая цифра $\frac{a}{b}$ отличается от $x = \frac{a_n a_{n-1}}{b_m b_{m-1}}$ не более чем на 1.

Доказательство. $\frac{a_n a_{n-1}}{b_m b_{m-1}+\frac{1}{base}} \leqslant \frac{a}{b} \leqslant \frac{a_n a_{n-1}+\frac{1}{base}}{b_m b_{m-1}} \Rightarrow |\frac{a}{b} - x| \leqslant (\frac{a_n a_{n-1}+\frac{1}{base}}{b_m b_{m-1}} - x) + (x - \frac{a_n a_{n-1}}{b_m b_{m-1}+\frac{1}{base}}) := y$

Заметим, $b_m \neq 0 \Rightarrow b_m b_{m-1} \geqslant base$. Продолжаем преобразования:
$y \leqslant \frac{1}{base} \cdot \frac{1}{b_m b_{m-1}} + \frac{a_n a_{n-1}}{base(b_m b_{m-1})^2} = \frac{1}{base \cdot b_m b_{m-1}}(1 + \frac{a_n a_{n-1}}{b_m b_{m-1}}) \leqslant \frac{1}{base^2}(1 + \frac{base^2}{base}) \leqslant 1.$ ■

• **Алгоритм деления:**

Длина частного, т.е. $\frac{n-m}{k}$, раз вычисляем α – приближение старшей цифры частного за $\mathcal{O}(1)$, затем умножением за $\mathcal{O}(\frac{n}{k})$ вычитаем $(\alpha-1)b\,10^{ki}$ из a и не более чем двумя вычитаниями $b\,10^{ki}$ доводим дело до конца. Важно было начать с $\alpha-1$, чтобы не уйти в минус при вычитании.

Лекция #7: Игры на графах

13 ноября 2017

7.1. Основные определения

Def 7.1.1. *Игра на орграфе: по вершинам графа перемещается фишка, за ход игрок должен сдвинуть фишку по одному из рёбер.*
Симметричная игра – *оба игрока могут ходить по всем рёбрам*
Несимметричная игра – *каждому игроку задано собственное множество рёбер*
Проигрывает игрок, который не может ходить.

Упражнение 7.1.2. Пусть по условию игры "*некоторые вершины являются выигрышными или проигрышными для некоторых игроков*". Такую игру можно вложить в определение выше.

Результат симметричной игры определяется графом G и начальной вершиной $v \in G$.
Игру будем обозначать (G, v), результат игры $r(G, v)$, или для краткости $r(v)$.
Классифицируем $v \in G$: в зависимости от $r(G, v)$ назовём вершину v
проигрышной (`L`), *выигрышной* (`W`) или *ничейной* (`D`).
Также введём множества: $\texttt{WIN} = \{v \mid r(v) = \texttt{W}\}$, $\texttt{LOSE} = \{v \mid r(v) = \texttt{L}\}$, $\texttt{DRAW} = \{v \mid r(v) = \texttt{D}\}$.

Замечание 7.1.3. В несимметричных играх, если вершина, например, проигрышна, ещё важно добавлять, какой игрок ходил первым: "проигрышная для 1-го игрока".

Lm 7.1.4. Если для вершины в условии задачи не указан явно её тип, то:

$$W \Leftrightarrow \exists \text{ ход в L}, \quad L \Leftrightarrow \text{ все ходы в W}$$

7.1.1. Решение для ациклического орграфа

Граф ацикличный $\Rightarrow$ вспомним про динамическое программирование.
`dp[v]` $= r(G, v)$; изначально "`-1`", т.е. "не посчитано".
База: пометим все вершины, информация о которых дана по условию.
Далее ленивая динамика:

```
int result( int v ) {
  int &r = dp[v];
  if (r != -1) return r;
  r = L; // например, если исходящих рёбер нет, результат уже верен
  for (int x : edges[v])
    if (result(x) == L) // ищем ребро в проигрышную
      r = W; // добавь break, будь оптимальнее!
  return r;
}
```

Замечание 7.1.5. Чтобы не придумывать ничего отдельно для несимметричных игр, обычно просто вводят новый граф, вершины которого – пары $\langle v, who \rangle$ (вершина и, кто ходит).

7.1.2. Решение для графа с циклами (ретроанализ)

Будем пользоваться 7.1.4. Цель: как только есть вершина, которая по лемме должна стать W/L, делаем её такой и делаем из этого некие выводы. Процесс можно реализовать через dfs/bfs. Мы выберем именно bfs, чтобы в будущем вычислить *длину игры*. Итак, ретроанализ:

```
queue <-- все вершины, которые по условию W/L.
while !queue.empty()
  v = queue.pop()
  for x in inner_edges[v]: // входящие в v рёбра
    if lose[v]:
      make_win(x, d[v]+1)
    else:
      if ++count[x] == deg[x]: // в count считается число проигрышных рёбер
        make_lose(x, d[v]+1)
```

Функции `make_win` и `make_lose` проверяют, что вершину помечают первый раз, если так, добавляют её в очередь. Второй параметр – обычное для `bfs` расстояние до вершины.
Все помеченные вершины по 7.1.4 помечены правильно.
Непомеченные вершины, чтобы им было не обидно, пометим D.

Теорема 7.1.6. Ничейные – ровно вершины с пометкой D.

Доказательство. Из вершины v типа D есть рёбра только в D и W (в L нет, т.к. тогда бы наш алгоритм пометил v, как W). Также из любой D есть хотя бы одно ребро в D. Мы игрок, мы находимся в D. Каков у нас выбор? Если пойдём в W, проиграем. Проигрывать не хотим $\Rightarrow$ пойдём в D $\Rightarrow$ вечно будем оставаться в D $\Rightarrow$ ничья. ■

Def 7.1.7. *$len(G, v)$ – длина игры, сколько ходов продлится игра, если выигрывающий игрок хочет выиграть максимально быстро, а проигрывающий максимально затянуть игру.*

Замечание 7.1.8. После ретроанализа `d[v]` $= len(G, v)$, так как ретроанализ:
1. Перебирал вершины в порядке возрастания расстояния.
2. Для выигрышной вершины брал наименьшую проигрышную.
3. Для проигрышной вершины брал наибольшую выигрышную.

Строго доказать можно по индукции. Инварианты: при обработке v все вершины со строго меньшей длиной игры уже обработаны; если посчитано $r(v)$, то посчитано верно.

Замечание 7.1.9. На практике разбиралась садистская версия той же задачи: проигрывающий хочет побыстрее выиграть и начать новую партию, а выигрывающий подольше наслаждаться превосходством (т.е. оставлять себе возможность выиграть). Описание решения: после первого ретроанализа поменять местами смысл L/W, запустить второй ретроанализ.

7.2. Ним и Гранди, прямая сумма

Def 7.2.1. *Прямая сумма графов* $G_1 = \langle V_1, E_1\rangle$ *и* $G_2 = \langle V_2, E_2\rangle$ *– граф с вершинами* $\langle v_1, v_2\rangle \,|\, v_1 \in V_1, v_2 \in V_2$ *и рёбрами* $(a,b) \to (a,c) \,|\, (b,c) \in E_2$ *и* $(a,b) \to (c,b) \,|\, (a,c) \in E_1$.

Def 7.2.2. *Прямая сумма игр:* $\langle G_1, v_1\rangle \times \langle G_2, v_2\rangle = \langle G_1 \times G_2, (v_1, v_2)\rangle$

По сути "у нас есть два графа, можно делать ход в любом одном из них".

Def 7.2.3. $mex(A) = \min x \mid x \geqslant 0, x \notin A$.

Пример 7.2.4. $A = \{0,1,7,10\} \Rightarrow mex(A) = 2; \quad A = \{1,2,3,4\} \Rightarrow mex(A) = 0$

Def 7.2.5. *На ациклическом орграфе можно задать Функцию Гранди* $f[v]$*:*
Пусть из v *ведут рёбра в* $Out(v) = \{x_1, x_2, \dots, x_k\} \overset{def}{\Rightarrow} f[v] = mex\{f[x_1], \cdots, f[x_k]\}$.

Lm 7.2.6. $f[v] = 0 \Leftrightarrow v \in$ `LOSE` (доказывается элементарной индукцией)

Пример 7.2.7. *Игра "спички".* На столе n спичек, за ход можно брать от 1 до 4 спичек. Кто берёт последнюю, проигрывает. Проверьте, что функция Гранди: $0,1,2,3,4,0,1,2,3,4,0,1,\dots$
При этом проигрывает тот, кто получает $n \vdots 5$, а чтобы выиграть, нужно взять $n \bmod 5$ спичек.

Пример 7.2.8. *Игра "Ним".* На столе n камней, за ход можно брать любое положительное число камней. Заметим $f[n] = n$, выигрышная стратегия – взять всё.

Пока теория не выглядит полезной. Чтобы осознать полезность, рассмотрим прямые суммы тех же игр – есть n кучек спичек или n кучек камней, и брать можно, соответственно, только из одной из кучек. На вопрос "кто выиграет в таком случае" отвечают следующие теоремы:

Теорема 7.2.9. $f[\langle v_1, v_2\rangle] = f[v_1] \oplus f[v_2]$

Доказательство. Для доказательства воспользуемся индукцией по размеру графа.
Из вершины v в процессе игры сможем прийти только в $C(v)$ – достижимые из v вершины.
Для любого ребра $\langle v_1, v_2\rangle \to \langle x_1, x_2\rangle$ верно, что $|C(\langle v_1, v_2\rangle)| > |C(\langle x_1, x_2\rangle)|$.
База индукции: или из v_1, или из v_2 нет рёбер.
Переход индукции: для всех $\langle G_1, x_1, G_2, x_2\rangle$ с $|C(\langle x_1, x_2\rangle)| < |C(\langle v_1, v_2\rangle)|$ уже доказали

$$f[\langle x_1, x_2\rangle] = f[x_1] \oplus f[x_2]$$

Осталось честно перечислить все рёбра из $\langle v_1, v_2\rangle$ и посчитать mex вершин, в которые они ведут.

$$A = \{\underbrace{f[v_1] \oplus f[x_{21}], f[v_1] \oplus f[x_{22}], \dots}_{\text{игрок сделал ход из } v_2}, \underbrace{f[x_{11}] \oplus f[v_2], f[x_{12}] \oplus f[v_2], \dots}_{\text{игрок сделал ход из } v_1}\}.$$

Здесь $Out(v_1) = \{x_{11}, x_{12}, \dots\}, Out(v_2) = \{x_{21}, x_{22}, \dots\}$. Доказываем, что $mex\, A = f[v_1] \oplus f[v_2]$.

Во-первых, поскольку $\forall i\, f[x_{1i}] \neq f[v_1] \wedge \forall i\, f[x_{2i}] \neq f[v_2]$, имеем $f[v_1] \oplus f[v_2] \notin A$.
Докажем, что все меньшие числа в A есть. Обозначим $x = f[v_1]$, $y = f[v_2]$, $M = f[v_1] \oplus f[v_2]$.
Будем пользоваться тем, что из v_1 есть ходы во все числа из $[0, x)$, аналогично $v_2 \to [0, y)$.
Пусть k – старший бит M и пришёл он из $x \Rightarrow$
ходами $x \to [0, 2^k)$ мы получим 2^k в x различных k-битных чисел, т.е. все числа из $[0, 2^k)$.
Чтобы получить числа $[2^k, M)$ перейдём от задачи $\langle x, y, M\rangle$ к $\langle x - 2^k, y, M - 2^k\rangle$.
Воспользуемся индукцией. База: $M = 0$. ■

Следствие 7.2.10. Для суммы большего числа игр аналогично: $f[v_1] \oplus f[v_2] \oplus \cdots \oplus f[v_k]$.

Замечание 7.2.11. Рассмотрим раскраску части плоскости $[0, +\infty) \times [0, +\infty)$.
В клетку $[i, j]$ ставится минимальное неотрицательное целое число, которого нет левее и ниже.
Получится ровно $i \oplus j$, так как мы ставим ровно mex в игре Ним на кучках $\{i, j\}$.

7.3. Вычисление функции Гранди

Ацикличный граф $\Rightarrow$ динамика. Осталось быстро научиться считать mex:
используем "обнуление" массива за $\mathcal{O}(1)$ и получим $\mathcal{O}(deg_v)$ на вычисление mex вершины v:

```
1 int cc, used[MAX_MEX]; // изначально нули
2 cc++;
3 for (int x : out_edges[v])
4   used[f[x]] = cc; // функция Гранди x уже посчитана
5 for (f[v] = 0; used[f[v]] == cc; f[v]++)
6   ;
```

Итого суммарное время работы $\mathcal{O}(V+E)$.
Нужно ещё оценить `MAX_MEX`. Тривиальная оценка даёт $\mathcal{O}(\max_v \deg_v)$, можно точнее:

Lm 7.3.1. $\forall G = \langle V, E \rangle,\ \forall v\ f[v] \leqslant \sqrt{2E}$

Доказательство. Пусть в графе есть вершина с функцией Гранди $k \Rightarrow$ из неё есть рёбра в вершины с функцией Гранди $0, 1, \ldots, k-1$. А из вершины с функцией Гранди $k-1$ есть ещё $k-1$ рёбер и т.д. Итого: $k + (k-1) + (k-2) + \cdots + 1 = \frac{k(k+1)}{2}$ рёбер $\Rightarrow k(k+1) \leqslant 2E \Rightarrow k \leqslant \sqrt{2E}$. ■

7.4. Эквивалентность игр

Напомним, игра на графе – пара $\langle G, v \rangle$. Материал главы относится к произвольным орграфам.

Def 7.4.1. *Игры A и B называются эквивалентными, если* $\forall C\ r(A \times C) = r(B \times C)$.

Def 7.4.2. *Игры A и B называются эквивалентными, если $A + B$ проигрышна.*

На лекции вам дано второе определение, обычно используют первое...
В любом случая важно понимать, что "эквивалентность" – отдельная глава, которой мы не пользовались, выводя фукнцию Гранди от прямой суммы игр.

TODO