Спортивное программирование

Алгоритмы и структуры данных

Версия от 14 мая 2023 г.

Этот документ доступен также в формате сайта.
Актуальную версию этого документа всегда можно найти по ссылке.
Если вы нашли опечатку или ошибку, сообщите, пожалуйста, мне в телеграме, если вам не лень: t.me/peltorator.
Прочитать про проект можно в блоге на codeforces.
С историей последних изменений можно ознакомиться по ссылке.

The Legrand Orange Book template

Оглавление

I	Запросы на отрезках

1 Префиксные суммы 5
1.1 Определения, основы и сила полуинтервалов 5
1.2 Базовое применение 6
1.3 А что кроме суммы? 7
1.4 Продвинутые применения 8
1.5 Двумерный случай 9
1.6 Трехмерный случай 11
1.7 Разностный массив 13
1.8 Применения разностного массива 13
1.9 Двумерный разностный массив 16
1.10 Задачи для практики 18

2 Стек рекордов (стек минимумов / максимумов) 19
2.1 Пример задачи 19
2.2 Определение, свойства и построение стека минимумов 21
2.2.1 Стек минимумов без стека минимумов 24
2.2.2 Один проход со стеком вместо двух 24
2.3 Применения 25
2.4 Источники 27
2.5 Задачи для практики 27

3 RMQ offline. Вариация алгоритма Тарьяна. 28
3.1 Задачи для практики 29

4 Sparse Table 30
4.1 Идея 30
4.2 Построение 31
4.3 Ответ на запрос 32
4.4 Применения 33
4.5 Разреженные таблицы для поиска суммы на отрезке 33
4.6 Многомерные разреженные таблицы 34
4.7 Задачи для практики 35

5 Дерево отрезков снизу 36
5.1 Изменение в точке, сумма на отрезке 36
5.2 Изменение на отрезке, значение в точке 39
5.3 Двумерные запросы 40
5.4 Ленивые обновления 41
5.5 Задачи для практики 43

6 Segment Tree Beats 45
6.1 Общая идея 46
6.2 %=, = в точке, $\sum$ 48
6.2.1 Формулировка 48
6.2.2 Решение 48
6.2.3 Доказательство 48
6.3 min=, $\sum$, max (Ji Driver Segment Tree) 50
6.3.1 Формулировка 50
6.3.2 Решение 50
6.3.3 Доказательство 51
6.4 min=, max=, +=, $\sum$, max, min 51
6.4.1 Формулировка 52
6.4.2 Решение 52
6.4.3 Доказательство 52
6.5 min=, +=, gcd 53
6.5.1 Формулировка 53
6.5.2 Решение + доказательство упрощенной версии задачи 54
6.5.3 Решение + доказательство полной версии задачи 55
6.6 %=, = на отрезке, $\sum$ 56
6.6.1 Формулировка 56
6.6.2 Решение 56
6.6.3 Доказательство 56
6.7 $\sqrt{}$=, +=, $\sum$, max, min 57
6.7.1 Формулировка 57
6.7.2 Решение 57
6.7.3 Доказательство 58

6.8 /=, +=, $\sum$, max, min 59
6.8.1 Формулировка . . . 59
6.8.2 Решение . . . 60
6.8.3 Доказательство . . . 60
6.9 &=, |=, max 61
6.9.1 Формулировка . . . 61
6.9.2 Решение . . . 61
6.9.3 Доказательство . . . 62
6.10 min=, +=, max over $\sum$ для нескольких массивов параллельно 62
6.10.1 Формулировка . . . 63
6.10.2 Решение + доказательство . . . 63
6.11 Историческая информация: количество изменений элемента 64
6.11.1 Формулировка . . . 64
6.11.2 Решение + доказательство . . . 64
6.12 Историческая информация: исторический максимум 64
6.12.1 Формулировка . . . 64
6.12.2 Решение + доказательство . . . 65
6.13 Историческая информация: историческая сумма 65
6.13.1 Формулировка . . . 65
6.13.2 Решение + доказательство . . . 65
6.14 Задачи для практики 66
7 Алгоритм Фараха-Колтона и Бендера . . . 67
7.1 RMQ offline. Вариация алгоритма Тарьяна за $O(\alpha(n))$ на запрос. 67
7.2 RMQ online. Улучшаем разреженные таблицы. 68
7.3 RMQ± 1 online. Маленьких отрезков становится мало. 68
7.4 Сведение LCA к RMQ± 1. 69
7.5 Сведение RMQ к LCA. 70
7.6 Упрощенная битовая вариация алгоритма ФКБ для практики 71
7.7 Задачи для практики 72
8 Дерево Ли Чао . . . 73
8.1 Мотивация 73
8.2 Идея 73
8.3 Удаление прямых 75
8.4 Кусочно-линейные функции 76
8.5 Ленивые обновления 77
8.6 Задачи для практики 77
II Деревья
9 Двоичные подъемы с линейной памятью . . . 80
9.1 Идея 80
9.2 Доказательство 81

9.3 Поиск предка на глубине h 82
9.4 Поиск наименьшего общего предка 82
9.5 Задачи для практики 83

III Динамическое программирование

10 Оптимизация Кнута — Яо 85
10.1 Алгоритм 85
10.1.1 Задача об оптимальном разбиении массива на k подотрезков 85
10.1.2 Динамика по подотрезкам 88
10.2 Достаточное условие для монотонности точки разреза 89
10.2.1 Достаточное условие для задачи оптимального разбиения на k подотрезков 89
10.2.2 Достаточное условие для задачи динамики по подотрезкам 89
10.3 Оптимизация Кнута — Яо для трудновычислимых стоимостей подотрезков разбиения 90
10.4 Источники 90
10.5 Задачи 90

11 Оптимизация разделяй-и-властвуй 91
11.1 Алгоритм 91
11.2 Достаточное условие для монотонности точки разреза 94
11.2.1 Лямбда-оптимизация и 1D1D — не панацея 94
11.3 Разделяй-и-властвуй для трудновычислимых стоимостей подотрезков разбиения 95
11.4 Источники 97
11.5 Задачи 97

12 Лямбда-оптимизация 98
12.1 Идея на примере задачи 98
12.2 Применение лямбда-оптимизации вне динамического программирования 104
12.3 Восстановление ответа 104
12.3.1 Хранение минимального и максимального количества подотрезков в оптимальных разбиениях 104
12.3.2 Второй бинпоиск для уточнения k 105
12.3.3 Комбинация двух путей 105
12.4 Доказательство выпуклости функции f 105
12.4.1 Сведение к mincost-k-flow 105
12.4.2 Неравенство четырехугольника 106
12.4.3 Задача линейного программирования 108
12.5 Источники 109
12.6 Задачи 110

13 Неравенство четырехугольника . . . 111
13.1 Формулировки решаемых задач 111
13.2 Формулировка неравенства четырехугольника 112
13.3 Ослабленное неравенство четырехугольника 114
13.4 Полезные следствия неравенства четырехугольника 115
13.5 Условие монотонности по включению 117
13.6 Источники 119

14 Персистентный Convex Hull Trick . . . 120
14.1 Задачи для практики 121

IV Теория чисел

15 Нахождение обратных ко всем остаткам за $O(p)$ . . . 123
15.1 Метод обратных факториалов 123
15.2 Алгоритм одного цикла 125

16 Поиск факториала по простому модулю . . . 127

17 Поиск факториала по простому модулю за $O(\sqrt{\min(p,n)}\log^2 n)$ 129
17.1 Задачи для практики 130

18 Обращение Мёбиуса, свертка Дирихле . . . 131
18.1 Формула обращения Мёбиуса 131
18.2 Свертка Дирихле 133
18.3 Поиск количества пар взаимнопростых, не больших n 134
18.4 Сумма попарных НОДов чисел, не больших n 137
18.5 Сумма попарных НОКов чисел, не больших n 137
18.6 Задачи для практики 139

19 Сумма мультипликативной функции: Powerful Number Sieve . 140

20 Квадратный корень по простому модулю за $O(\log p)$ . . . 143

21 Дискретное логарифмирование . . . 145

22 Оценка на количество делителей числа и сверхсоставные числа 146
22.1 Предыстория и мотивация 146
22.2 Субполиномиальность и сверхсоставные числа 146
22.3 Быстрая генерация больших сверхсоставных чисел 147
22.4 Таблица для повседневного использования 148
22.5 Численный анализ оценки в кубический корень 148

22.6 Более точная практическая оценка 150

V Алгебра

23 Сумма по подмножествам и xor-and-or-свертки . . . 153
23.1 Сумма по подмножествам 153
23.1.1 Обращение суммы по подмножествам . . . 154
23.1.2 Сумма по надмножествам . . . 155
23.1.3 Сумма по подмножествам для проверки леммы Холла . . . 156
23.2 or-свертка и and-свертка 157
23.2.1 and-свертка . . . 158
23.3 xor-свертка 159
23.3.1 Преобразование Уолша — Адамара и его применение . . . 159
23.3.2 Вычисление преобразования Уолша — Адамара и его обращения . . . 160
23.3.3 Альтернативный математический взгляд . . . 162
23.4 Источники 162
23.5 Задачи для практики 162

VI Геометрия

24 Поиск пары ближайших точек за $O(n)$ . . . 165
24.1 Задачи для практики 167
25 Проверка пересечения полуплоскостей на непустоту за $O(n)$ . . . 168
26 Минимальная покрывающая окружность за $O(n)$ . . . 171
27 Поиск пересечения полуплоскостей с точкой внутри . . . 173

VII C++, среды, стрессы, тактика и стратегия

28 Генерация случайных чисел . . . 176
28.1 mt19937 176
28.2 uniform_int_distrubition 178
28.3 Как генерировать случайные числа по модулю 179
28.4 Другие распределения 179
28.5 Выбор сида рандома 179
29 Стресс-тестирование . . . 181
29.1 Общие концепции 181
29.2 Linux: Shell стрессы 183
29.3 Windows: Bat стрессы 185
29.4 Python стрессы 185

29.5 In-Code стрессы 186

30 Быстрый ввод-вывод в C++ 187
30.1 endl и \n 187
30.2 sync_with_stdio и cin.tie 188
30.3 Ручной быстрый ввод-вывод 188

VIII Дополнительные материалы

I Запросы на отрезках

1 Префиксные суммы 5

2 Стек рекордов (стек минимумов / максимумов) 19

3 RMQ offline. Вариация алгоритма Тарьяна. 28

4 Sparse Table 30

5 Дерево отрезков снизу 36

6 Segment Tree Beats 45

7 Алгоритм Фараха-Колтона и Бендера 67

8 Дерево Ли Чао 73

1. Префиксные суммы

1.1 Определения, основы и сила полуинтервалов

Определение 1.1.1 Префиксными суммами массива $[a_0, a_1, a_2, \ldots, a_{n-1}]$ называется массив $[b_0, b_1, b_2, \ldots, b_n]$, определяющийся следующим образом:

$$
\begin{aligned}
&b_0 = 0\\
&b_1 = a_0\\
&b_2 = a_0 + a_1\\
&b_3 = a_0 + a_1 + a_2\\
&\ldots\\
&b_{n-1} = a_0 + a_1 + \ldots + a_{n-2}\\
&b_n = a_0 + a_1 + \ldots + a_{n-1}
\end{aligned}
$$

Замечание 1.1.1 Сила полуинтервалов
Обратите внимание на то, что b_i — это сумма первых i элементов массива a. Иногда префиксные суммы определяют так, что $b_i = a_0 + a_1 + \ldots + a_i$, но этот способ неудобен на практике, в чем мы убедимся далее.

На данном примере можно познакомиться с очень важной концепцией в алгоритмах: практически всегда вместо отрезков лучше использовать полуинтервалы. К примеру, в данном случае b_i — это сумма элементов массива a на полуинтервале $[0, i)$, что на практике окажется удобнее, чем хранить в b_i сумму на отрезке $[0, i]$.

Замечание 1.1.2 Также стоит помнить о том, что длина массива b на один больше длины массива a.

Замечание 1.1.3 Формулу для b_i можно записать рекуррентно:

$$b_0 = 0$$
$$b_{i+1} = b_i + a_i, \quad \text{где} \quad i \geqslant 0$$

Из рекуррентной формулы сразу становится ясно, как посчитать массив префиксных сумм за $O(n)$:

```
vector<int> build_prefix_sums(const vector<int>& a) {
    int n = a.size();
    vector<int> prefix_sums(n + 1, 0);
    for (int i = 0; i < n; i++) {
        prefix_sums[i + 1] = prefix_sums[i] + a[i];
    }
    return prefix_sums;
}
```

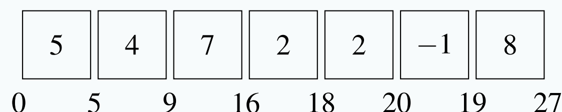
Замечание 1.1.4 Обратите внимание, что элементы массива префиксных сумм — это суммы большого количества элементов исходного массива, поэтому будьте аккуратнее с переполнением. И вообще, на протяжении всей этой темы вы можете столкнуться с переполнением, поэтому будьте всегда начеку! Возможно, вам нужен тип `long long` вместо `int`.

Кроме того, есть встроенная в C++ функция `std::partial_sum`, которая как раз таки считает префиксные суммы. Пример ее работы:

```
vector<int> build_prefix_sums(const vector<int>& a) {
    int n = a.size();
    vector<int> prefix_sums(n + 1, 0);
    partial_sum(a.begin(), a.end(), prefix_sums.begin() + 1);
    return prefix_sums;
}
```

Обратите внимание, что сама функция `partial_sum` не оставляет нуля в начале, поэтому нам приходится делать это самим, добавляя единицу к `prefix_sums.begin()`.

Пояснение 1.1 У нас уже есть две интуиции для понимания b_i: сумма первых i элементов исходного массива и сумма элементов исходного массива на полуинтервале $[0, i)$. Давайте посмотрим на еще один вариант того, как об этом можно думать. Можно представить, что элементы массива находятся в ячейках, а префиксные суммы находятся между ними — на перегородках. И содержат в себе суммы всего того, что находится перед этой перегородкой.



1.2 Базовое применение

Давайте сразу же применим префиксные суммы на примере задачи.

Задача 1.1 Дан массив целых чисел, и приходят запросы вида «найти сумму на полуинтервале с позиции l до позиции r». Нужно отвечать на запросы за $O(1)$.

> **Решение 1.1** Давайте изначально перед ответами на запросы предпосчитаем массив префиксных сумм. Тогда если бы во всех запросах l было равно нулю, то ответом на запрос была бы просто префиксная сумма b_r.
>
> Но как же действовать, если $l \neq 0$? В префиксной сумме b_r содержатся все нужные нам элементы, однако есть еще лишние: $a_0, a_1, \ldots, a_{l-1}$. Но ведь сумма этих элементов — это как раз таки b_l. Таким образом, выполнено тождество:
>
> $$a_l + a_{l+1} + \ldots + a_{r-1} = b_r - b_l$$
>
> То есть для ответа на запрос поиска суммы на полуинтервале нужно просто вычесть друг из друга две предпосчитанные префиксные суммы. ■

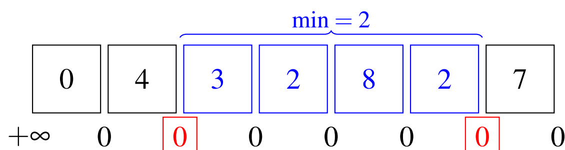
```
1 int get_sum(int left, int right) { // [left, right)
2     return b[right] - b[left];
3 }
```

> **Замечание 1.2.1** Обратите внимание на то, какие красивые формулы у нас получаются: сумма на полуинтервале $[l, r)$ — это $b_r - b_l$. Такая красота достигается именно благодаря тому, что мы используем полуинтервалы: и запросы у нас даны в виде полуинтервалов, и префиксные суммы. Если бы префиксные суммы были посчитаны в виде $b_i = a_0 + a_1 + \ldots + a_i$, то в формуле появились бы неприятные ± 1, в которых легко запутаться: $b_r - b_{l-1}$, и случай, когда $l = 0$, стал бы крайним, его надо было бы разбирать отдельно.

1.3 А что кроме суммы?

Давайте зададимся вопросом: для каких операций можно использовать префиксные суммы? Не только же для сложения? Какими свойствами должна обладать операция, чтобы можно было воспользоваться префиксными суммами?

На самом деле, необходимо, чтобы функция, которую мы считаем на отрезке, была обратима, что равносильно тому, что должна быть возможность по двум префиксам восстановить значение на отрезке. К примеру, операция суммы обратима, потому что если мы прибавили лишнее, потом это можно вычесть. А операции минимума и максимума необратимы. Нельзя по значениям минимумов на префиксах получить значение минимума на отрезке. К примеру, если элемент на позиции 0 в массиве самый маленький, то все префиксные минимумы будут равны этому элементу, но минимумы на каких-то отрезках совсем с ним не связаны.



Но кроме суммы есть и другие операции, которые являются обратимыми. Одна из самых популярных — это, пожалуй, операция «побитового исключающего или»[1], которая еще называется «xor» и обозначается $\oplus$.

[1] Подробнее про эту операцию можно прочитать здесь

При этом для `xor`'а пользоваться префиксными суммами еще удобнее. Выполнено тождество $x \oplus x = 0$ для любого числа x, что означает, что операция `xor` обратна сама себе, так что формула вычисления побитового исключающего или на отрезке получается такая:

$$a_l \oplus a_{l+1} \oplus \ldots \oplus a_{r-1} = b_r \oplus b_l$$

1.4 Продвинутые применения

Давайте решим еще пару задач, в которых нам понадобятся префиксные суммы.

Задача 1.2 Дан массив. Необходимо за $O(n \log n)$ найти любой его непустой подотрезок с нулевой суммой элементов.

> **Решение 1.2** Как мы уже знаем, суммы на отрезках — это разности префиксных сумм. Поэтому то, что сумма на отрезке равна нулю, равносильно тому, что префиксные суммы его концов равны.
>
> Таким образом, мы свели задачу нахождения подотрезка нулевой суммы к задаче нахождения двух одинаковых элементов в массиве префиксных сумм. Для этого можно, к примеру, отсортировать массив префиксных сумм и искать совпадающие элементы среди соседних. Либо же можно воспользоваться хеш-таблицей (`unordered_map` в C++), и тогда асимптотика решения будет вовсе $O(n)$. ■

> **Упражнение 1.1** Даны два массива одинаковой длины. Необходимо найти такой подотрезок, чтобы сумма элементов первого массива на этом подотрезке совпадала с суммой элементов второго массива на этом подотрезке. Асимптотика $O(n)$. ■

Задача 1.3 Кузнечик находится в клетке с индексом 0 и хочет попасть в клетку с индексом n. За один прыжок кузнечик может переместиться на любое количество клеток вперед, но при этом не меньше l и не больше r. Найдите, сколько существует маршрутов кузнечика. Асимптотика $O(n)$.

> **Решение 1.3** Задача очень похожа на стандартную задачу о кузнечике, однако в ней прыжки имели длину 1 и 2, а теперь у нас количество разных прыжков не ограничено, поэтому такое же решение будет работать за $O(n \cdot (r-l))$, что в худшем случае будет $O(n^2)$.
>
> Давайте внимательно посмотрим на формулу пересчета динамики. Количество способов попасть в клетку i — это сумма количеств способов попасть во все предыдущие клетки на пути кузнечика:
>
> $$dp_i = dp_{i-r} + dp_{i-r+1} + \ldots + dp_{i-l-1} + dp_{i-l}$$
>
> То есть элемент массива dp определяется через сумму отрезка элементов того же массива. Давайте идти по позициям в порядке возрастания и не только насчитывать массив dp, но и его префиксные суммы. Тогда пересчет dp_i через префиксные суммы будет работать за $O(1)$, а асимптотика всего алгоритма — $O(n)$.
>
> Таким образом, мы видим, что в задачах необязательно предпосчитывать массив префиксных сумм заранее, он может строиться по ходу решения задачи и одновременно с этим использоваться. ■

> **Замечание 1.4.1** Префиксные суммы очень удобны для подсчета суммы на отрезке в том случае, если массив в ходе запросов не меняется. Потому если какой-то элемент массива поменялся, то нужно пересчитать все префиксные суммы, в которые он входит. Это очень долго! Если есть запросы изменения, то лучше подойдут более продвинутые структуры данных: к примеру, дерево отрезков и дерево Фенвика.

1.5 Двумерный случай

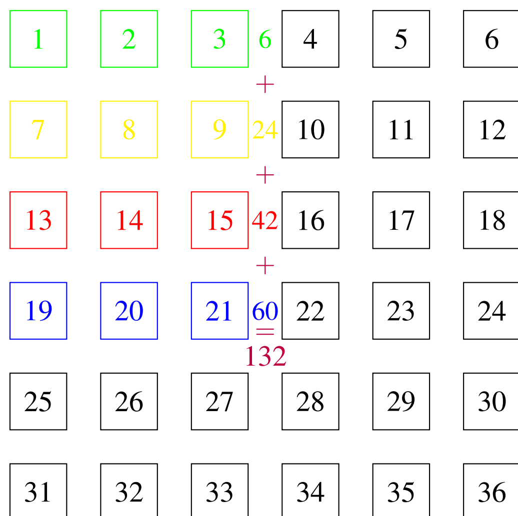
Мы уже научились искать сумму на отрезке в одномерном массиве при помощи префиксных сумм, но префиксные суммы легко обобщаются и на бо́льшие размерности. Давайте научимся искать сумму на прямоугольнике в двумерном массиве. Пусть нам дан двумерный массив $a_{i,j}$. Определим его префиксные суммы так:

$$b_{i+1,j+1} = \sum_{x \leqslant i} \sum_{y \leqslant j} a_{x,y} = a_{0,0} + a_{0,1} + a_{0,2} + \ldots + a_{0,j} + a_{1,0} + a_{1,1} + \ldots + a_{1,j} + \ldots +$$

$$+ a_{i,0} + a_{i,1} + \ldots + a_{i,j}$$

Для двумерных префиксных сумм тоже полезно представлять, что они находятся между значениями исходного массива в узлах сетки и отвечают за сумму всего, что выше и левее этой точки.

Можно воспринимать происходящее, как будто мы построили префиксные суммы по одной координате, а затем посчитали префиксные суммы префиксных сумм по другой координате.



Таким образом, можно насчитать префиксные суммы на одномерных массивах, а потом насчитать префиксные суммы на массивах префиксных сумм.

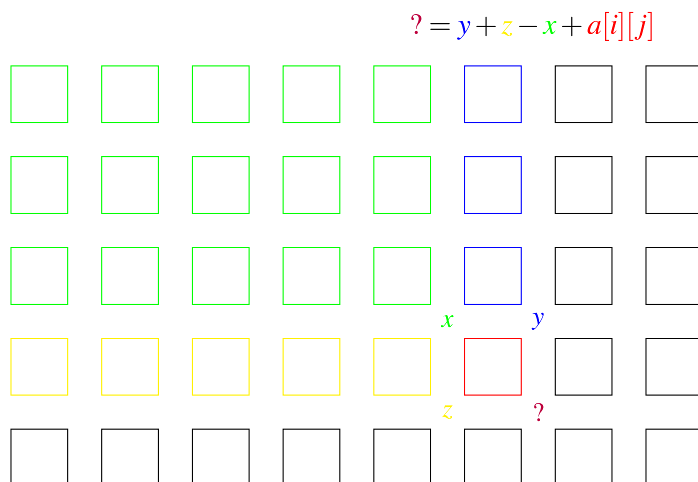
```
1 vector<vector<int>> build_prefix_sums_2d(const
     vector<vector<int>>& a) {
2     int n = a.size();
3     int m = a[0].size();
4     vector<vector<int>> prefix_sum_1d(n);
5     for (int i = 0; i < n; i++) {
6         prefix_sum_1d[i] = build_prefix_sums(a[i]);
7     }
```

```
8      vector<vector<int>> prefix_sum_2d(n + 1, vector<int>(m +
          1, 0));
9      for (int j = 0; j <= m; j++) {
10         for (int i = 0; i < n; i++) {
11             prefix_sum_2d[i + 1][j] = prefix_sum_2d[i][j] +
                  prefix_sum_1d[i][j];
12         }
13     }
14     return prefix_sum_2d;
15 }
```

Альтернативный вариант подсчета префиксных сумм — вновь воспользоваться рекуррентной формулой. Это будет немного сложнее, чем в одномерном случае. Формула выглядит следующим образом:

$$b_{i+1,j+1} = b_{i,j+1} + b_{i+1,j} - b_{i,j} + a_{i,j}$$

Эту формулу легко понять из картинки (зеленая область также принадлежит синей и желтой):



Мы берем сумму двух меньших префиксных сумм, которые накрывают нашу, однако их пересечение учтется дважды, поэтому его надо вычесть. Но ведь это пересечение — это и есть $b_{i,j}$. И в конце стоит не забыть прибавить новый элемент — $a_{i,j}$.

Элементы, через которые мы считаем $b_{i+1,j+1}$ имеют меньшие индексы, поэтому подсчет двумерных префиксных сумм можно вести просто двумя вложенными циклами по возрастанию.

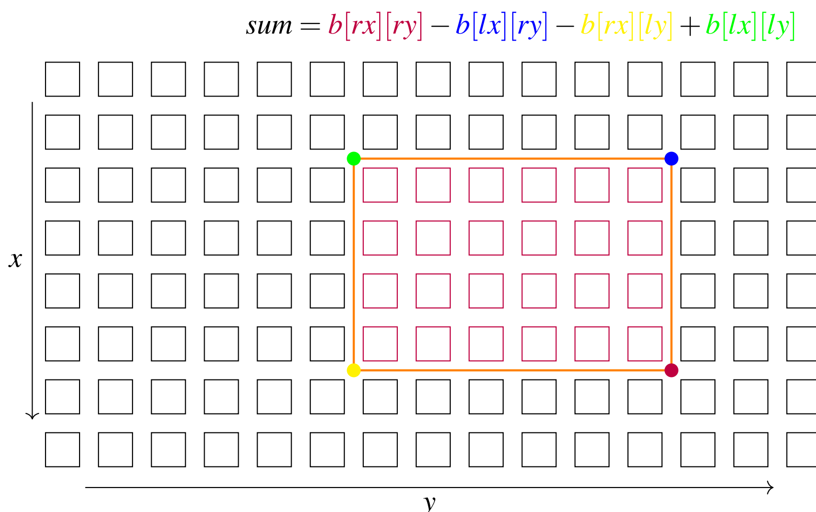
```
1  vector<vector<int>> build_prefix_sums_2d(const
      vector<vector<int>>& a) {
2      int n = a.size();
3      int m = a[0].size();
4      vector<vector<int>> prefix_sum(n + 1, vector<int>(m + 1,
          0));
5      for (int i = 0; i < n; i++) {
6          for (int j = 0; j < m; j++) {
```

```
7               prefix_sum[i + 1][j + 1] = prefix_sum[i][j + 1] +
                   prefix_sum[i + 1][j] - prefix_sum[i][j] +
                   a[i][j];
8           }
9       }
10      return prefix_sum;
11  }
```

Теперь пускай нам надо найти сумму на «полупрямоугольнике» (двумерный полуинтервал) с противоположными углами в точках (lx, ly) и (rx, ry) (левые границы — включительно, правые — исключительно). Сумма элементов на этом полупрямоугольнике будет выражаться так:

$$\sum_{lx \leqslant i < rx} \sum_{ly \leqslant j < ry} a_{i,j} = b_{rx,ry} - b_{lx,ry} - b_{rx,ly} + b_{lx,ly}$$

Эту формулу также легче понимать, смотря на картинку:



Сначала мы взяли большой прямоугольник, который накрывает нужный нам, потом удалили ненужное слева и сверху, но то, что находится на их пересечении, мы удалили дважды, так что надо вернуть это пересечение назад.

> **Замечание 1.5.1** Именно в двумерном случае, а также больших размерностях становится видна мощь полуинтервалов. В формуле нет ни одной цифры (± 1). Мы просто берем префиксные суммы на краях и складываем и вычитаем их.
>
> Если бы мы определяли $b_{i,j}$ как сумму на отрезке, то есть $\sum_{x \leqslant i} \sum_{y \leqslant j} a_{x,y}$, то формула для суммы на прямоугольнике выглядела бы следующим образом:
>
> $$b_{rx,ry} - b_{lx-1,ry} - b_{rx,ly-1} + b_{lx-1,ly-1}$$
>
> Вероятность написать эти формулы правильно постепенно стремится к нулю.

1.6 Трехмерный случай

Мы разобрались с двумерным случаем, теперь пойдем еще дальше! Префиксные суммы на трехмерном массиве определяются аналогично:

$$b_{i+1,j+1,k+1} = \sum_{x \leqslant i} \sum_{y \leqslant j} \sum_{z \leqslant k} a_{x,y,z}$$

Рекурентная формула для их подсчета будет такая:

$$b_{i+1,j+1,k+1} = b_{i,j+1,k+1} + b_{i+1,j,k+1} + b_{i+1,j+1,k} - b_{i+1,j,k} - b_{i,j+1,k} - b_{i,j,k+1} + b_{i,j,k} + a_{i,j,k}$$

И для поиска суммы на «полупараллелепипеде» (трехмерном полуинтервале) с противоположными углами в точках (lx, ly, lz) и (rx, ry, rz) подойдет следующая формула:

$$\sum_{lx \leqslant i < rx} \sum_{ly \leqslant j < ry} \sum_{lz \leqslant k < rz} a_{i,j,k} = b_{rx,ry,rz} - b_{lx,ry,rz} - b_{rx,ly,rz} - b_{rx,ry,lz} + b_{rx,ly,lz} + b_{lx,ry,lz} + b_{lx,ly,rz} - b_{lx,ly,lz}$$

Формулы страшные! Но показываю я вам их для того, чтобы сформулировать правила, по которым они работают в общем случае для любой размерности. Потому что если у вас есть пространственное мышление, то по картинке придумать формулу для трехмерного случая вы еще сможете, а вот для четырехмерного уже вряд ли.

Теорема 1.6.1 В рекуррентной формуле подсчета префиксных сумм мы берем комбинацию всех возможных вариантов вычитания единиц из индексов. При этом если мы вычли нечетное количество единиц, то слагаемое идет с плюсом, а если четное, то с минусом. В конце нужно не забыть еще прибавить один новый элемент изначального массива a.

Понять, почему именно нечетное с плюсом, можно по тому, что мы точно знаем, что слагаемые, в которых всего из одной координаты вычли единицу, должны идти с плюсом, ведь при их помощи мы накрываем изначально нашу префиксную сумму, а потом уже остальными пытаемся ее корректировать.

Теорема 1.6.2 Аналогичное правило есть и для формулы поиска суммы на полупараллелепипеде. Берется сумма всевозможных комбинаций левых и правых границ по каждой координате, при этом если левых границ четное количество, то слагаемое берется с плюсом, а если нечетное, то с минусом.

Опять же, понять, почему именно четное с плюсом, а нечетное с минусом, очень легко: слагаемое, в котором все границы правые — это самое большое слагаемое, которое мы потом корректируем, так что оно точно должно быть с плюсом.

Упражнение 1.2 При помощи этих двух незатейливых правил попробуйте составить формулы для построения префиксных сумм и поиска суммы на (полу)гиперпрямоугольнике в четырехмерном пространстве. ■

Замечание 1.6.3 Трехмерный префиксный массив можно насчитать другим способом так же как и двумерный: сначала посчитать одномерные префиксные суммы, потом на них насчитать двумерные и затем уже на них посчитать трехмерные.

Замечание 1.6.4 Если вы знаете, что такое формула включений-исключений, то в каком-то смысле именно она и применяется при построении и при ответе на запросы в многомерных префиксных суммах.

1.7 Разностный массив

Мы уже научились по массиву строить его массив префиксных сумм. А теперь давайте научимся по массиву префиксных сумм строить исходный массив, то есть применять обратную операцию.

Определение 1.7.1 Разностным массивом массива $[b_0, b_1, \ldots, b_{n-1}]$ называется массив $[a_0, a_1, \ldots, a_{n-2}]$, определяющийся следующим образом:

$$
\begin{aligned}
a_0 &= b_1 - b_0\\
a_1 &= b_2 - b_1\\
a_2 &= b_3 - b_2\\
&\ldots\\
a_{n-3} &= b_{n-2} - b_{n-3}\\
a_{n-2} &= b_{n-1} - b_{n-2}
\end{aligned}
$$

Очевидно, что если b — массив префиксных сумм массива a, то массив a — разностный массив массива b, потому что формула $a_i = b_{i+1} - b_i$ — это просто преобразованная рекуррентная формула для поиска префиксных сумм: $b_{i+1} = b_i + a_i$. Однако разностный массив может помочь нам даже там, где массивом префиксных сумм и не пахнет! Также обратите внимание, что если для подсчета массива префиксных сумм была нужна рекуррентная формула, то каждый член разностного массива зависит всего от двух элементов исходного, так что можно пользоваться формулами из определения для подсчета разностного массива за $O(n)$.

```
1 vector<int> build_diffs_array(const vector<int>& arr) {
2     int n = arr.size();
3     vector<int> diffs(n - 1);
4     for (int i = 0; i < n - 1; i++) {
5         diffs[i] = arr[i + 1] - arr[i];
6     }
7     return diffs;
8 }
```

Замечание 1.7.1 Если вы знакомы с основами матанализа, можно заметить, что переход к разностному массиву — это дискретное дифференцирование, а переход к массиву префиксных сумм — дискретное интегрирование.

1.8 Применения разностного массива

Решим несколько задач, используя разностный массив. Мы уже решили задачу нахождения суммы на отрезке при помощи префиксных сумм, а теперь рассмотрим в некотором смысле обратную задачу: задачу о прибавлении на отрезке.

Задача 1.4 Дан массив длины n. Приходят q запросов: прибавить на полуинтервале $[l, r)$ ко всем элементам число d. После выполнения всех запросов необходимо вывести

получившийся массив. Асимптотика $O(n+q)$.

Решение 1.4 В замечании 1.4.1 мы говорили о том, что если элементы массива меняются, то для решения задачи понадобятся продвинутые структуры, такие как дерево отрезков или дерево Фенвика. Однако здесь мы обойдемся без них, потому что у нас есть только запросы изменения, а запрос «получения» есть только один в самом конце.

Давайте в начало исходного массива b допишем фиктивный элемент ноль и у получившегося массива возьмем разностный массив a. После чего будем наблюдать за тем, как этот массив a будет меняться в следствии запросов изменения массива b на отрезке. Фиктивный ноль мы добавили для того, чтобы не потерять информацию о нулевом элементе массива b, ведь только по разностям соседних элементов восстановить исходный массив не получится.

Пускай на полуинтервале $[l,r)$ исходного массива (или на полуинтервале $[l+1,r+1)$ массива с фиктивным нулем в начале) прибавили ко всем элементам d. Заметим, что элементы массива a на позициях меньших l и больших r никак не поменяются, потому что оба элемента в разности никак не поменялись. На позициях $l+1,\dots,r-1$ тоже ничего не поменяется, потому что к обоим элементам разности прибавят d, в результате чего сама разность не изменится. К примеру, для позиции $l+1$:

$$b_{l+1}^{new} - b_l^{new} = \left(b_{l+1}^{old} + d\right) - \left(b_l^{old} + d\right) = b_{l+1}^{old} + d - b_l^{old} - d = b_{l+1}^{old} - b_l^{old}$$

Таким образом, после операции прибавления на отрезке изменятся только два элемента разностного массива: a_l заменится на a_l+d, и a_r заменится на a_r-d.

Алгоритм получается следующий: изначально перейдем от исходного массива к его разностному массиву, предварительно добавив в начало массива 0, чтобы не потерять информацию о b_0. Затем выполняем операции изменения за $O(1)$ каждую, потому что нужно поменять каждый раз всего два элемента. И в конце нужно вернуться к исходному виду массива, посчитав префиксные суммы. ■

```
1  vector<int> precalc(vector<int> b) {
2      b.insert(b.begin(), 0); // add leading zero
3      vector<int> a = build_diffs_array(b);
4      return a;
5  }
6
7  void add_on_half_interval(int l, int r, int d) { // [l, r) +=
       d
8      a[l] += d;
9      if (r < n) {
10         a[r] -= d;
11     }
12 }
13
14 vector<int> postcalc(const vector<int>& a) {
15     vector<int> finalb = build_prefix_sums(a);
16     finalb.erase(finalb.begin()); // delete leading zero
17     return finalb;
18 }
```

Замечание 1.8.1 Обратите внимание на то, что если $r = n$, то такого элемента нет в массиве a, ведь этот элемент отвечал бы за разность элемента после конца массива b с последним элементом массива b, которая нас не интересует. Поэтому в таком случае мы ничего не делаем.

Замечание 1.8.2 На самом деле можно было даже не брать разностный массив. Можно было считать, что изначально массив состоял из всех нулей, тогда его разностный массив тоже состоит из всех нулей; на этом массиве произвести все операции, посчитать массив префиксных сумм и уже в самом конце прибавить начальные значения элементов массива.

Задача 1.5 Дан массив длины n. Приходят q запросов: прибавить на полуинтервале $[l, r)$ арифметическую прогрессию с шагом $step$, то есть к элементу на позиции l прибавить $step$, к элементу на позиции $l+1$ прибавить $2 \cdot step$, к элементу на позиции $l+2$ прибавить $3 \cdot step$, ..., и наконец к элементу на позиции $r-1$ прибавить $(r-l) \cdot step$.

После выполнения всех запросов необходимо вывести получившийся массив. Асимптотика $O(n+q)$.

Решение 1.5 Задача похожа на предыдущую, но явно сложнее, ведь к каждому элементу на отрезке прибавляется разное число. Давайте посмотрим, что произойдет, если мы как и раньше перейдем к разностному массиву. Заметим, что если мы на полуинтервале прибавили $step, 2 \cdot step, \ldots, (r-l) \cdot step$, то в разностном массиве мы на некотором полуинтервале прибавим ко всем элементам $step$, а также из элемента на правой границе вычтется $(r-l) \cdot step$. Но ведь прибавлять число на отрезке мы уже умеем! Давайте перейдем к разностному массиву разностного массива, при этом не забыв в таком случае добавить уже не один, а два фиктивных нуля в начало массива. Тогда для разностного массива разностного массива изменятся только элементы на позициях l, r и $r+1$, так что мы можем выполнять все операции за $O(1)$, а затем в конце дважды насчитать массив префиксных сумм, вернувшись к исходному массиву. ■

```
1  vector<int> precalc(vector<int> b) {
2      b.insert(b.begin(), 0);
3      b.insert(b.begin(), 0); // add two leading zeros
4      vector<int> a = build_diffs_array(build_diffs_array(b));
5      return a;
6  }
7
8  // [l, r) += [step, 2 * step, ..., (r - l) * step]
9  void add_arithm_on_half_interval(int l, int r, int step) {
10     a[l] += step;
11     if (r < n) {
12         a[r] -= (r - l + 1) * step;
13     }
14     if (r + 1 < n) {
15         a[r + 1] += (r - l) * step;
16     }
17 }
18
19 vector<int> postcalc(const vector<int>& a) {
```

```
20     vector<int> finalb =
          build_prefix_sums(build_prefix_sums(a));
21     finalb.erase(finalb.begin());
22     finalb.erase(finalb.begin()); // delete leading zeros
23     return finalb;
24 }
```

Упражнение 1.3 Попробуйте решить последнюю задачу, если первый член арифметической прогрессии не совпадает с шагом, то есть на полуинтервале прибавляются числа $start, start + step, start + 2 \cdot step, \ldots, start + (r - l - 1) \cdot step$. ■

Упражнение 1.4 Как обобщить решение прошлой задачи на случай, когда на отрезке прибавляется не линейная, а квадратичная функция? То есть прибавляются числа $step, 4 \cdot step, 9 \cdot step, \ldots, (r - l)^2 \cdot step$. ■

Упражнение 1.5 Представьте, что у вас есть список операций виде «прибавить на полуинтервале $[l, r)$ значение d». Но вам нужно выполнять не эти операции, а целые отрезки операций! То есть в действительности операция имеет вид «примените к массиву операции с номерами от L-й до R-й». В конце нужно вывести получившийся массив. Как решать такую задачу за $O(n + q)$? ■

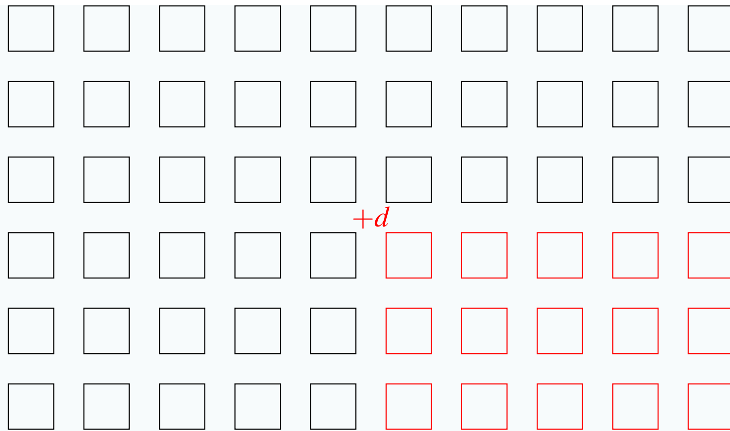
Замечание 1.8.3 Если вы знаете основы матанализа, можно легко понять, почему мы производили именно такие манипуляции. Ведь если взять производную, то прибавление константы на отрезке превратится в прибавление тождественного нуля, а для линейной функции нужно взять вторую производную.

1.9 Двумерный разностный массив

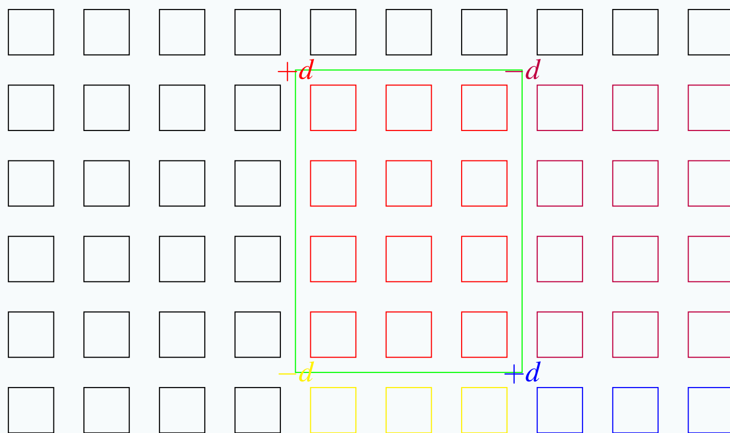
Задача 1.6 Дан двумерный массив размера $n \times m$, изначально состоящий из всех нулей. Приходит q запросов прибавления константы на прямоугольнике. В конце нужно вывести элементы получившегося массива.

Решение 1.6 Это двумерная версия задачи, которую мы рассматривали ранее, но для простоты изначальный массив состоит из нулей, поэтому нам не нужно переходить к разностному массиву. Надо подумать, к каким элементам что надо прибавить на разностном массиве, чтобы на исходном массиве прибавить d на полупрямоугольнике $[lx, rx) \times [ly, ry)$.

Можно заметить, что если в разностном массиве к какому-то элементу прибавить d, то в исходном массиве d прибавится ко всем элементам на **суффиксном** подпрямоугольнике.



Теперь перед нами стоит задача «прибавить на подпрямоугольнике через прибавления на суффиксных подпрямоугольниках». Эта задача аналогична задаче поиска суммы на подпрямоугольнике.



Прибавляем мы к тем же самым клеткам, сумму которых мы брали в задаче поиска суммы на подпрямоугольнике.

В конце мы возьмем префиксные суммы получившегося двумерного массива, и это и будет ответом.

```
// [lx, rx) * [ly, ry) += d
void add_on_half_rectangle(int lx, int ly, int rx, int ry,
   int d) {
    diffs[lx][ly] += d;
    if (ry < m) {
        diffs[lx][ry] -= d;
    }
    if (rx < n) {
        diffs[rx][ly] -= d;
    }
    if (rx < n && ry < m) {
        diffs[rx][ry] += d;
    }
}

vector<vector<int>> postcalc(const vector<vector<int>>&
```

```
        diffs) {
16        vector<vector<int>> final_arr =
              build_prefix_sums_2d(diffs);
17        final_arr.erase(final_arr.begin());
18        // delete leading horizontal zeros
19        for (size_t i = 0; i < final_arr.size(); i++) {
20            final_arr[i].erase(final_arr[i].begin());
21            // delete leading vertical zeros
22        }
23        return final_arr;
24    }
```

Замечание 1.9.1 Как и раньше, мы добавляем нули в начало. В случае двумерного массива мы добавили строку и столбец нулей. В качестве упражнения остается проверить, что индексы, к которым надо прибавлять, будут именно такие. При этом как и в одномерном случае, правые индексы могут не существовать в массиве `diffs`, если они равны высоте или ширине изначального массива.

Упражнение 1.6 Спросите себя: понимаете ли вы, что делать, если изначально массив состоит не из нулей? ■

Упражнение 1.7 Как обобщить решение предыдущей задачи, если массив уже не двумерный, а трехмерный, и прибавлять надо на параллелепипеде? ■

Определение 1.9.1 Строго говоря, разностный двумерный массив можно строить так же, как и одномерный. Формула для его элементов будет такая:

$$a_{i+1,j+1} = b_{i+1,j+1} - b_{i,j+1} - b_{i+1,j} + b_{i,j}$$

что похоже на аналогичную формулу для двумерных префиксных сумм. Но как вы видели, на практике нам не надо изначально строить разностный массив. Нужно просто изначально считать, что он заполнен нулями, а после выполнения операций по нему посчитать массив префиксных сумм. И начальные элементы добавить уже в самом конце.

1.10 Задачи для практики

- Предлагается прорешать специально подготовленный контест на codeforces. Если у вас нет доступа к соревнованию, нужно сначала вступить в группу.

2. Стек рекордов (стек минимумов / максимумов)

TL;DR **Стек минимумов** (максимумов) — это элементы массива, правее которых до текущего индекса i нет меньших (бо́льших) элементов. Часто используется вместо деревьев отрезков и других более сложных структур данных в задачах, где необходимо понимать структуру массива относительно запросов поиска минимума или максимума на отрезке. Строится последовательно для индексов по возрастанию: чтобы из стека минимумов (максимумов) для индекса $i-1$ получить стек минимумов (максимумов) для индекса i, нужно удалить из стека все индексы, значения которых не меньше (не больше) a_i, а затем добавить индекс i в конец стека. Такое построение суммарно занимает $O(n)$ времени. Для большего понимания обратитесь к картинке и коду ниже.

Стек рекордов или стек минимумов или максимумов (не путать со стеком **с** минимумом) — очень простая, но при этом очень мощная структура данных, которая во многих задачах может заменить такие более сложные структуры данных как дерево отрезков. При этом в силу ее простоты, отдельное время во время изучения алгоритмов ей не уделяется, поэтому в интернете сложно найти какие-либо материалы по этой теме. На английском языке стек рекордов известен под названием «monotonic stack».

2.1 Пример задачи

Рассмотрим стандартный пример задачи, в которой можно применить стек рекордов.
Задача 2.1 Дан массив целых чисел a длины n. Необходимо найти сумму минимумов по всем его подотрезкам или другими словами $\sum_{l=0}^{n-1}\sum_{r=l}^{n-1}\min(a_l, a_{l+1}, \ldots, a_r)$.

Очевидным решением за $O(n^3)$ будет перебрать все отрезки и для каждого из них найти минимум. Если зафиксировать левую границу отрезка и перебирать правую по возрастанию, можно поддерживать минимум на лету. Такое решение будет работать за $O(n^2)$. Однако, чтобы продолжить улучшать асимптотику, необходимо сменить перспективу.

Заметим, что минимум на любом отрезке равен какому-то конкретному элементу на этом отрезке (а именно, минимуму). Для простоты пока предположим, что все элементы массива различны. Тогда зафиксируем для каждого отрезка такой элемент. Нам необходимо найти сумму всех таких чисел. Давайте поменяем точку зрения: вместо того чтобы для каждого отрезка рассматривать минимум, мы для каждого элемента a_i массива посчитаем значение cnt_i: для скольки отрезков он является минимумом. Тогда ответом на задачу будет $\sum_{i=0}^{n-1} a_i \cdot cnt_i$. Нам остается лишь найти эти значения cnt_i. Но в каком случае a_i является минимумом на каком-то отрезке $[l,r]$? Во-первых, элемент a_i должен лежать на этом отрезке, а во-вторых, все остальные элементы на отрезке должны быть больше a_i. В частности, все элементы на отрезке $[l,i-1]$ больше a_i и все элементы на отрезке $[i+1,r]$ больше a_i. И если это так, то любой такой отрезок $[l,r]$ нам подходит. Тогда предположим, что l_0 — это ближайший индекс слева от i, такой что $a_{l_0} < a_i$, и r_0 — ближайший справа от i такой индекс. Тогда очевидно, что l должно быть больше l_0, потому что иначе отрезок $[l,i+1]$ будет содержать l_0, а также r должно быть меньше r_0, потому что иначе отрезок $[i+1,r]$ будет содержать r_0. При этом если $l_0 < l \leqslant i \leqslant r < r_0$, то любой такой отрезок нам подходит, ведь он содержит индекс i и при этом не содержит никаких элементов, меньших a_i. Существует $i-l_0$ подходящих левых границ и r_0-i подходящих правых границ, и нам подходит любая такая пара, так что $cnt_i = (i-l_0)\cdot(r_0-i)$. Остается лишь для каждого индекса i найти ближайшие слева и справа элементы, которые меньше a_i. В этот момент вам, возможно, хочется сказать «наверное, это как-то делается через дерево отрезков», но не стоит попадать в эту ловушку! Стек минимумов — это как раз таки ровно структура данных, которая позволяет для каждого элемента массива найти ближайший меньший элемент всего в несколько строчек кода без использования каких-либо сложных структур данных типа дерева отрезков.

Но перед тем как мы перейдем непосредственно к рассмотрению стека минимумов, сначала мы должны исправить пару неаккуратностей, которые мы допустили в алгоритме выше. Во-первых, что делать, если таких индексов l_0 или r_0 не существует? Что, если нет ни одного элемента левее a_i, который был бы его меньше? В таком случае в качестве левой границы мы можем выбрать любой индекс от 0 до i. Иными словами, можно считать, что $l_0 = -1$. Аналогично для правой границы можно считать, что $r_0 = n$. Это можно понимать немного иначе, представив, что в массиве есть дополнительные элементы a_{-1} и a_n, которые равны $-\infty$, и поэтому у каждого элемента массива всегда есть элементы слева и справа, которые меньше. Вторая проблема — это то, что мы считали, что все элементы массива различны. Если в массиве есть одинаковые элементы, то на отрезках может быть несколько элементов, равных минимуму, и нам нужно точно определить, какой из них мы выберем, чтобы случайно не посчитать один и тот же отрезок несколько раз. Не умаляя общности, можно считать, что мы всегда выбираем самый правый среди минимумов на отрезке (можно было выбрать самый левый; главное — зафиксировать выбор и следовать ему). Это значит, что все элементы на отрезке правее него строго меньше его, а все элементы на отрезке левее него меньше или равны ему. То есть на самом деле нам надо найти не ближайшие слева и справа элементы, меньшие данного, а слева найти ближайший меньший, а справа ближайший не больший. Кардинально это никак алгоритм не поменяет. Альтернативный способ избавиться от проблемы равных чисел — это представить, что массив состоит не из элементов a_i, а из пар (a_i, i), которые сравниваются лексикографически (сначала по первому элементу, а при равенстве первого элемента, по второму). В таком случае все элементы массива точно будут различны, так что минимум на любом отрезке будет единственен. На самом деле это в точности соответствует нашему предыдущему

способу, если бы среди равных чисел мы выбирали самое левое, потому что если на отрезке есть несколько элементов с одинаковым значением, то меньшим среди них будет элемент с минимальным индексом.

2.2 Определение, свойства и построение стека минимумов

Теперь перейдем непосредственно к стеку минимумов, и как он может помочь нам найти для каждого индекса массива ближайший слева меньший элемент. Для начала определим, что такое стек минимумов.

> **Определение 2.2.1** Стек минимумов (стек рекордов) для индекса i массива a — это возрастающая последовательность индексов массива a, такая что индекс $j \leqslant i$ находится в этой последовательности тогда и только тогда, когда a_j строго меньше всех остальных элементов на отрезке $[j,i]$. Если же вместо строго меньше в предыдущем предложении написать строго больше, то такая последовательность называется стеком максимумов.

В частности, последним элементом стека минимумов для индекса i всегда является индекс i, потому что a_i больше всех остальных элементов на отрезке $[i,i]$, потому что это единственный элемент на этом отрезке. Первым же элементом в стеке минимумов индекса i является индекс минимума на отрезке $[0,i]$ (самый правый, если есть несколько элементов, равных минимуму), потому что он удовлетворяет условию на вхождение в стек минимумов, но никакой элемент левее него не может удовлетворять этому условию, иначе он был бы меньше минимума. В зависимости от контекста для удобства иногда считают, что первым элементом стека минимумов является -1 (подразумевая, что в начале массива есть элемент $a_{-1} = -\infty$).

Почему же эта структура называется стеком рекордов? Если мы встанем в точку i и пойдем налево, то элементы стека минимумов будут в точности «рекордными» значениями, которые меньше всего, что мы видели до этого. В частности, из этого легко сделать вывод, что не только индексы в стеке минимумов строго возрастают, но и соответствующие им значения в массиве a. Также это значит, что если j и k — последовательные элементы стека рекордов индекса i, то элемент a_k является минимумом (самым правым минимумом, если в массиве могут быть одинаковые элементы) на отрезках с правой границей i тогда и только тогда, когда левая граница такого отрезка l строго больше j и не больше k ($j < l \leqslant k$). Эта идея является ключевой в одном из главных алгоритмов, использующем стек рекордов, — вариации алгоритма Тарьяна для поиска минимума на отрезке в оффлайне (Глава 3). Тем самым весь массив до индекса i имеет очень конкретную структуру относительно элементов стека минимумов: элементы стека минимумов — это строго возрастающая последовательность элементов, а все элементы между двумя последовательными элементами стека минимумов не меньше их обоих. Если нарисовать элементы массива как точки (i,a_i) на плоскости, то мы также сможем получить и красивую визуальную геометрическую интуицию (см. Рис. 2.1).

Кроме того, если мы обозначим стек минимумов для индекса i как последовательность j_0, j_1, ..., j_m, то, как мы уже поняли, $j_0 = -1$ и $j_m = i$, но кроме того, для любого индекса j_k в стеке минимумов, j_{k-1} является для него тем самым ближайшим слева меньшим элементом (это становится очевидно, если вспомнить процесс, когда мы встаем в индекс i и идем налево в поиске «рекордов»). И эта идея как раз таки позволяет нам найти то, что мы хотели, — ближайший слева от i индекс l, такой что $a_l < a_i$, ведь если у нас уже есть стек минимумов, то этот индекс l — это просто предпоследний элемент стека минимумов j_{m-1}.

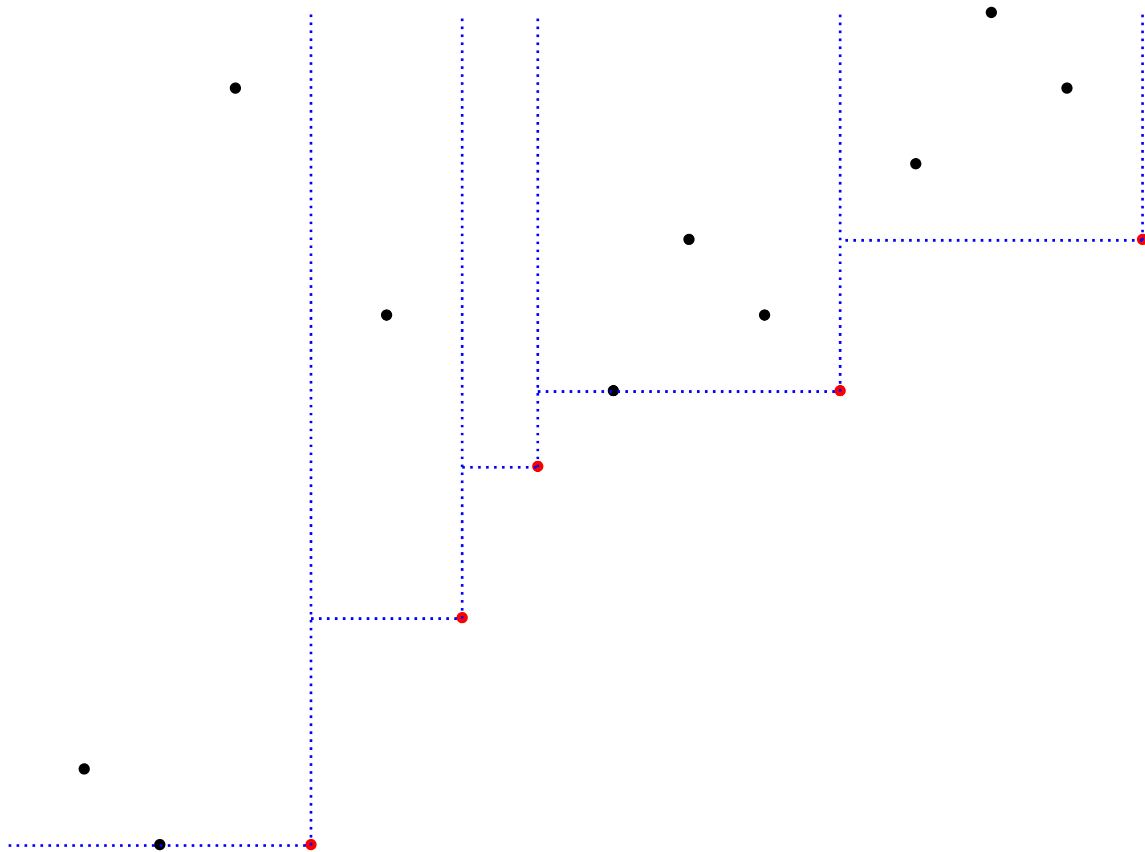


Рис. 2.1: Здесь красные точки — это элементы стека минимумов, а черные — все остальные. Обратите внимание, что все остальные элементы между двумя элементами стека минимумов не меньше их обоих, поэтому все элементы лежат внутри синих стаканов.

Мы обсудили основные свойства стека минимумов, и, надеюсь, теперь вы согласны, что он имеет фундаментальное значение для понимания структуры массива. В частности, он поможет нам решить задачу поиска суммы минимумов по всем подотрезкам массива. Последний шаг, который нам остается, — это, собственно, найти стеки минимумов для всех индексов массива. Обратите внимание, что сохранить их все у нас вряд ли получится, потому что, к примеру, если массив a возрастающий, то стек минимумов для индекса i содержит все индексы от -1 до i, поэтому суммарный размер всех стеков минимумов будет квадратичным относительно размера массива. Мы же сделаем нечто немного другое: мы будем перебирать индексы массива a по возрастанию и преобразовывать стек минимумов для индекса $i-1$ в стек минимумов для индекса i шаг за шагом. Таким образом, в момент, когда мы находимся в индексе i, перед нами стек минимумов для индекса i, который мы можем использовать.

Как же поменяется стек минимумов, если мы перейдем от индекса $i-1$ к индексу i? Вспомним определение: стек минимумов — это последовательность всех индексов j, таких что a_j меньше всех других элементов на отрезке от j до $i-1$. Если какой-то элемент a_j не был меньше всех элементов правее него, то при добавлении элемента a_i он таковым не станет. А если элемент j был в стеке минимумов, то он уже точно меньше всех элементов от j до $i-1$, так что он должен остаться в стеке минимумов тогда и только тогда, когда он меньше a_i. Так как мы уже знаем, что значения в массиве a элементов стека минимумов возрастают, то все индексы j, которые не удовлетворяют этому условию, находятся на конце стека. То есть алгоритм пересчета стека минимумов следующий: мы удаляем индексы j с вершины стека, пока $a_j \geqslant a_i$, после чего, когда такие элементы закончились, добавляем индекс i на вершину стека. И это все! На самом деле все очень просто. Надеюсь, теперь стало понятно, почему такую последовательность мы назвали стеком. Для большего понимания алгоритма рекомендую посмотреть на код:

```
1  // a -- array of length n
2  vector<int> stack_of_min;
3  for (int i = 0; i < n; i++) {
4      while (!stack_of_min.empty() && a[stack_of_min.back()] >=
           a[i]) {
5          stack_of_min.pop_back();
6      }
7      stack_of_min.push_back(i);
8      // use stack_of_min as a stack of minimums for index i
9  }
```

Такой алгоритм работает за линейное время, потому что каждый индекс массива добавляется в стек один раз, а значит, удалить из стека рекордов мы можем не более n элементов за все время.

Вы можете обратить внимание, что в коде сверху мы используем `std::vector` вместо стека. Это связано с тем, что в более продвинутых контекстах нам может хотеться использовать не только последний элемент этого стека, но и иметь доступ к другим элементам. В зависимости от ситуации, как мы уже обсуждали, бывает полезно добавить изначально в стек элемент -1, и тогда проверка `!stack_of_min.empty()` превращается в `stack_of_min.back() != -1`. В частности, это может помочь избежать разбора случаев в задаче поиска суммы минимумов по всем подотрезкам массива, которую мы, собственно, уже почти решили. Мы научились искать для каждого элемента массива ближайший меньшей слева (это вершина стека перед добавлением

индекса i в конец). Теперь, чтобы найти ближайший справа меньший или равный, нужно запустить аналогичный цикл, но по убыванию индексов (положив изначально n в стек), и удалять элементы, пока они **строго** больше a_i (легко запомнить, что мы удаляем элементы, пока они не удовлетворяют необходимому нам условию: если нам нужен ближайший меньший, мы удаляем, пока он не меньше, если нам нужен ближайший меньший или равный, мы удаляем, пока он больше, если нам нужен ближайшие больший, мы удаляем, пока он не больше, если нам нужен ближайший больший или равный, мы удаляем, пока он меньше). Когда мы нашли те самые нужные индексы слева и справа, ответ на задачу выражается через них простой формулой, как мы уже говорили ранее. С полным решением задачи можно ознакомиться по ссылке.

> **Замечание 2.2.1** Если мы кроме прохода со стеком минимумов еще сохраним для каждого индекса i значение pr_i (индекс ближайшего слева меньшего элемента), то чтобы позже восстановить стек минимумов индекса i, нужно рассмотреть индексы i, pr_i, pr_{pr_i}, $pr_{pr_{pr_i}}$, ... Тем самым, в случае, когда нам нужно иметь доступ к стеку минимумов в онлайне, мы можем воспользоваться идеей из персистентного Convex Hull Trick (Глава 14). Но в реальности в подавляющем числе случаев оффлайн прохода со стеком достаточно.

2.2.1 Стек минимумов без стека минимумов

Если нам нужно просто найти ближайший слева меньший элемент, то на самом деле явным образом хранить стек минимумов необязательно, ведь, как мы знаем, он как раз таки состоит из элементов i, pr_i, pr_{pr_i}, $pr_{pr_{pr_i}}$ и так далее. В таком случае удаление элемента с вершины стека равноценно переходу по ссылке pr к следующему элементу. Такое решение выражается в следующий код:

```
1 // a -- array of length n
2 vector<int> pr(n, -1);
3 for (int i = 0; i < n; i++) {
4     pr[i] = i - 1;
5     while (pr[i] >= 0 && a[pr[i]] >= a[i]) {
6         pr[i] = pr[pr[i]];
7     }
8 }
```

Этот код на самом деле делает все те же самые операции, что и решение со стеком, поэтому тоже работает за линейное время. Однако нам не надо поддерживать стек, и код стал чище. По духу это решение похоже на алгоритм поиска префикс-функции.

2.2.2 Один проход со стеком вместо двух

Для простоты предположим, что все элементы массива различны. Тогда пока мы ищем ближайший слева элемент, меньший текущего, и удаляем элементы со стека, мы можем одновременно искать ближайший больший удаляемых элементов. Им будет наш текущий элемент, ведь мы как раз удаляем элемент с вершины стека ровно в тот момент, когда мы нашли меньший элемент.

```
1 // a -- array of length n
2 vector<int> stack_of_min;
3 vector<int> pr_l(n, -1); // closest smaller to the left
4 vector<int> pr_r(n, n); // closest smaller or equal to the
     right
```

```
 5 for (int i = 0; i < n; i++) {
 6     while (!stack_of_min.empty() && a[stack_of_min.back()] >=
          a[i]) {
 7         pr_r[stack_of_min.back()] = i;
 8         stack_of_min.pop_back();
 9     }
10     if (!stack_of_min.empty()) {
11         pr_l[i] = stack_of_min.back();
12     }
13     stack_of_min.push_back(i);
14 }
```

В случае, если элементы в массиве могут совпадать, данный код найдет ближайший слева меньший и ближайший справа меньший или равный элементы массива. Для того чтобы и справа найти ближайший меньший, придется хранить дополнительную информацию в стеке. Но на самом деле, как и в примере с поиском суммы минимумов по всем подотрезкам, в большинстве задач мы как раз таки и хотим найти ближайший слева меньший, а справа меньший или равный.

Описанные выше два подхода можно объединить в один код, который без стека находит ближайший слева меньший и ближайший справа меньший или равный:

```
 1 // a -- array of length n
 2 vector<int> pr_l(n, -1); // closest smaller to the left
 3 vector<int> pr_r(n, n); // closest smaller or equal to the
      right
 4 for (int i = 0; i < n; i++) {
 5     pr_l[i] = i - 1;
 6     while (pr_l[i] >= 0 && a[pr_l[i]] >= a[i]) {
 7         pr_r[pr_l[i]] = i;
 8         pr_l[i] = pr_l[pr_l[i]];
 9     }
10 }
```

2.3 Применения

Мы уже обсудили, как стек минимумов может помочь нам решить задачу поиска суммы минимумов по всем подотрезкам массива за линейное время. Кроме того, его можно применить для ответа на запросы поиска минимума на отрезке в оффлайн (Глава 3). Далее мы приведем еще несколько примеров задач, в которых стек рекордов может заменить более сложные структуры данных.

Задача 2.2 Дан массив $a_0, a_1, \ldots, a_{n-1}$. Необходимо для каждого индекса i найти количество индексов j, таких что a_j не меньше всех элементов, расположенных не дальше от a_i чем a_j. Иными словами, количество индексов j, таких что не существует индекса k, такого что $|k-i| \leqslant |j-i|$ и $a_k > a_j$.

Заметим, что для индекса $i = n-1$ такие индексы j — это в точности элементы стека нестрогих минимумов, потому что для индекса $n-1$ все другие индексы находятся слева, а условие на то, что все ближайшие элементы не меньше данного, как раз эквивалентно условию нахождения в стеке минимумов. Однако для других индексов все не так просто, потому что нас волнуют не только индексы слева, но и справа. Давайте снова перевернем задачу. Вместо того чтобы для индекса i смотреть на все

подходящие индексы j, мы для каждого индекса j найдем, для каких индексов i он подходит. Во-первых, на отрезке между i и j не должно быть никаких элементов, бо́льших a_j. То есть, к примеру, i должен быть не левее ближайшего слева числа, большего a_j, а также не правее ближайшего справа числа, большего a_j. Обозначим таким позиции за l_0 и r_0, как и раньше. Тогда на самом деле, если есть такой индекс k, который ломает желаемые нам условия, то он должен быть равен либо l_0, либо r_0, потому что все остальные индексы только дальше. Тогда нам надо найти такие индексы i, чтобы для них j был ближайшим индексом среди j, l_0 и r_0. Легко понять, что i подходит тогда и только тогда, когда он лежит на отрезке $\left[\left\lceil\frac{l_0+j+1}{2}\right\rceil, \left\lfloor\frac{j+r_0-1}{2}\right\rfloor\right]$. То есть каждый индекс j добавляет 1 к ответу для какого-то отрезка индексов i, который мы можем легко посчитать, если посчитаем l_0 и r_0 для всех индексов, используя проходы со стеком максимумов слева направо и справа налево. Теперь нам лишь остается для каждого индекса i понять, сколько отрезков через него проходят. Это можно без труда сделать, используя разностный массив и префиксные суммы (подробнее в Главе 1). Таким образом, мы решили эту задачу за линейное время, избавившись от использования деревьев отрезков дважды: благодаря стеку максимумов и благодаря разностному массиву.

Задача 2.3 Дано число L и два массива положительных целых чисел h_i и w_i. Необходимо посчитать значения динамики

$$dp_i = \min_{j<i, w_{j+1}+w_{j+2}+\ldots+w_i \leqslant L} dp_j + \max(h_{j+1}, h_{j+2}, \ldots, h_i)$$

Это задача с USACO 2012 US Open, Gold Division. Там представлена мотивация для такой странной динамики через легенду про организацию книг в полки с ограничением на ширину, минимизируя суммарную высоту полок.

Как же решать такую задачу? Заметим, что если мы можем упаковать n книг с суммарной высотой полок H, то с такой же высотой мы можем упаковать и меньшее количество книг, поэтому dp не убывает. Тогда если для двух индексов j и k значения максимума на отрезках $(j,i]$ и $(k,i]$ равны, то оптимально будет пересчитать динамику через меньший из этих двух индексов. Но максимум на отрезке $(j,i]$ — это как раз таки ближайший справа от j элемент стека максимумов для индекса i. И если j_1 и j_2 — два последовательных индекса в стеке максимумов, таких что через них можно пересчитывать динамику, то среди всех индексов на $[j_1, j_2-1]$ оптимально пересчитывать динамику именно через j_1, потому что для них всех минимум на отрезке до i равен h_{j_2}. Таким образом, практически всегда выгодно пересчитывать динамику через индексы из стека максимумов кроме одной ситуации: если через очередной элемент стека максимумов уже нельзя пересчитать динамику, потому что сумма ширин книг больше L. В таком случае мы можем использовать метод двух указателей, чтобы всегда поддерживать самый первый индекс k, через который мы все еще можем пересчитываться (используя массив префиксных сумм w или поддерживая сумму ширин на отрезке $(k,i]$). Тогда мы будем поддерживать стек максимумов не как стек, а как дек, удаляя элементы из его начала, если через них уже нельзя пересчитываться, а также рядом со стеком максимумов хранить `std::set` значений $dp_j + h_{j'}$ для каждого элемента стека максимумов j, где j' — следующий элемент стека максимумов. Такие значения легко пересчитывать при удалении и добавлении элементов в стек. Тогда для того, чтобы посчитать значение dp_i, достаточно взять минимум из минимума значений сета и пересчета через индекс k, который мы поддерживаем двумя указателями. Итоговая асимптотика — $O(n \log n)$.

И на самом деле, подобные динамики встречаются во многих других задачах. Ниже даны только несколько примеров задач, которые решаются абсолютно таким же методом. И в целом, стек рекордов — это очень простая и одновременно с этим очень мощная структура данных, которая может сэкономить вам написание дерева отрезков и лишний логарифм в асимптотике во многих задачах.

2.4 Источники

- Статья о том, что можно использовать стек минимумов без стека минимумов.
- Пост про параллельный подсчет ближайшего меньшего слева и справа.

2.5 Задачи для практики

- Задача на поиск суммы минимумов по всем подотрезкам массива.
- Задача с USACO 2012 US Open, Gold Division, разобранная выше.
- Задача (спойлеры к петрозаводским сборам), аналогичная последней разобранной задаче выше.
- Еще одна задача, аналогичная последней разобранной задаче выше.
- Задача, в которой применяется идея, схожая с построением одновременно ближайших слева и справа меньших.
- Коллекция задач на стек минимумов.

3. RMQ offline. Вариация алгоритма Тарьяна.

Общеизвестен Алгоритм Тарьяна[1] для поиска *LCA* offline. Можно построить аналогичный алгоритм для решения задачи *RMQ*.

Так как задача нам дана в оффлайне, мы можем отвечать на запросы в любом порядке. Давайте для каждого индекса массива сохраним все запросы, для которых этот индекс является правой границей. После чего пройдемся по массиву слева направо, постепенно отвечая на запросы для текущей правой границы. При этом будем еще поддерживать стек минимумов[2] (Глава 2).

Заметим, что стек минимумов делит массив на отрезки: каждый элемент стека минимумов является минимумом на любом отрезке с левой границей от текущего элемента стека минимумов до следующего влево. Тогда чтобы найти минимум на отрезке, достаточно для левой границы отрезка найти ближайший справа элемент стека минимумов, он и будет являться минимумом на отрезке.

Как же мы будем находить этот самый ближайший элемент в стеке минимумов? Это можно делать при помощи бинпоиска по стеку, однако в таком решении асимптотика будет $O(n+m\log n)$, что нас не устраивает. Давайте поддерживать СНМ (систему непересекающихся множеств), в которой элементами будут являться индексы массива, а множествами — отрезки, высекаемые стеком минимумов, на которых минимум фиксирован. Тогда если мы в корне каждого дерева СНМ будем еще поддерживать минимум на множестве, то для данной левой границы достаточно дойти до корня ее дерева и там узнать ответ. Обратите внимание, что множества объединяются очень простым образом. Когда к нам пришел новый элемент, он сначала находится один в своем множестве, но когда мы удаляем со стека элемент, мы объединяем соответствующее множество с множеством текущей правой границы. Каждый элемент будет добавлен в стек (а следовательно, и удален) не более одного раза, так что мы

[1] Обратите внимание, что в статье по ссылке написано, что алгоритм работает за $O(n+m)$, что не совсем верно. Однако из-за того, что обратная функция Аккермана не больше 5 для любых практически возможных ограничений ($n, m \leqslant 10^{10^{10^{19500}}}$), можно считать, что это константа. Не смотря на это, в данной статье мы попытаемся соблюдать формальность.

[2] Не путать со стеком с минимумом!

сделаем $O(n)$ операций с СНМом. Асимптотика получается равной $O((n+q)\alpha(n))$, если использовать и эвристику сжатия путей, и ранговую эвристику.

Реализация доступна по ссылке.

Наиболее эффективная реализация доступна по ссылке.

3.1 Задачи для практики

- Задача на поиск максимума на отрезке.
- ОСТОРОЖНО! СПОЙЛЕРЫ К РОИ!
  Эту задачу можно очень просто сдать на высокий балл при помощи предложенного алгоритма.

4. Sparse Table

Sparse Table или разреженная таблица — это простая структура данных, которая позволяет за $O(1)$ отвечать на запросы поиска минимума и максимума на отрезке в неизменяющемся массиве. Минусом является то, что на ее построение уходит $O(n \log n)$ времени и памяти.

4.1 Идея

Давайте рассматривать эту структуру на примере задачи поиска минимума на отрезке (RMQ). Для операции максимума все будет аналогично.

Идея этой структуры кроется в ее названии: разреженная таблица. Что же это за таблица, и почему она разреженная? Под таблицей здесь понимается таблица, в которой хранятся ответы на все возможные запросы поиска минимума на отрезке. То есть квадрат, в котором по вертикали отложены все возможные левые границы, а по горизонтали все возможные правые границы. Имея такую таблицу, можно без труда отвечать на запросы поиска минимума на отрезке просто обращаясь к соответствующим полям нашей таблицы. Однако чтобы построить такую таблицу, нужно потратить $O(n^2)$ времени и памяти, что абсолютно недопустимо.

В этот момент мы решаем, что хранить всю таблицу мы не будем, а будем хранить разреженную таблицу. То есть сохраним значения не для всех возможных отрезков, а только для некоторых. Однако это должны быть такие отрезки, чтобы из ответов для них можно было сконструировать ответ для любого отрезка. В обычной таблице любой отрезок уже был предпосчитан. Такого мы себе позволить не можем, но давайте сделаем немного более слабое условие: должно выполняться условие, что любой отрезок массива можно накрыть двумя предпосчитанными отрезками. Тогда минимум на отрезке — это минимум из двух минимумов на этих подотрезках. При этом эти подотрезки могут пересекаться. В этот момент как раз таки мы и воспользуемся тем, что мы ищем именно минимум на отрезке, а не сумму. Если бы мы искали сумму, то нам было бы необходимо, чтобы подотрезки, на которые мы разбиваем наш отрезок, не пересекались, иначе пересечение мы учтем дважды.

Для минимума это непринципиально: если мы возьмем дважды минимум с одним и тем же числом, то результат не поменяется. Такое свойство операций называется идемпотентностью. Другие идемпотентные операции на отрезке тоже можно считать при помощи разреженных таблиц: максимум (аналогично минимуму), НОД[1] ($gcd(a,a) = a$) и т.д.

Какие же отрезки нужно предпосчитать, чтобы любой отрезок запроса разбивался на два предпосчитанных отрезка, но при этом нам не нужно было предпосчитывать очень много всего? Давайте предпосчитаем минимум для всех отрезков, длины которых равны степени двойки. Сколько всего таких отрезков? Для каждой длины существует не более n отрезков такой длины (при фиксированной длине у всех таких отрезков разные левые границы). При этом разных степеней двойки, не больших n, есть $\lfloor \log n \rfloor$ штук: 1, 2, 4, ..., $2^{\lfloor \log n \rfloor}$. Так что наша структура будет занимать $O(n \log n)$ памяти.

О том, как эту структуру построить и хранить, мы поговорим позже, а пока поймем, почему теперь любой отрезок разбивается на два предпосчитанных подотрезка. Пускай длина отрезка равна len. Пусть 2^k — это максимальная степень двойки, не бóльшая len. Тогда давайте возьмем один подотрезок длины 2^k, начинающийся в левой границе отрезка, а другой — заканчивающийся в правой границе. Очевидно, что они будут полностью лежать внутри отрезка, потому что $2^k \leqslant len$, осталось понять, почему они полностью покрывают отрезок, то есть пересекаются. Действительно, если они не пересекаются, то внутри отрезка длины len можно поместить два непересекающихся отрезка длины 2^k, тогда длина отрезка не меньше, чем $2^k + 2^k = 2^{k+1}$, но ведь мы взяли число 2^k так, чтобы это была максимальная степень двойки, не бóльшая len. Противоречие.

4.2 Построение

Теперь поговорим про то, как мы будем строить и хранить эту структуру. Хранить мы ее будем в двумерном массиве размера $\log n \times n$. Для каждой степени двойки и для каждой левой границы сохраним минимум на соответствующем отрезке. Важно обратить внимание на то, что не для любой левой границы и степени двойки существует соответствующий отрезок. К примеру, для последнего индекса массива есть отрезок только длины 1. У этой проблемы есть два решения: первое — просто игнорировать такие отрезки, потому что они нам все равно не понадобятся, второе — если отрезок вылезает за пределы массива, обрезать его концом массива. Конечно, не делать что-то легче, чем делать, поэтому мы выберем первый вариант, потому что эти отрезки все равно не используются.

Теперь нужно понять, как быстро насчитать нашу разреженную таблицу. Заметим, что для построения можно воспользоваться идеей, похожей на идею поиска минимума на любом отрезке разбиением его на два маленьких. Действительно, любой отрезок длины 2^k — это два отрезка длины 2^{k-1}, поэтому нужно просто взять минимум из двух минимумов. Тогда если мы изначально определим, что минимумы на отрезках длины 1 — это просто элементы изначального массива, а потом будем считать нашу таблицу по возрастанию степени двойки, то каждое значение можно будет посчитать за $O(1)$ по следующей формуле:

```
sparse[k][ind] = min(sparse[k - 1][ind], sparse[k - 1][ind + (1 << (k - 1))])
```

Нам нужно найти минимум на полуинтервале $[ind, ind + 2^k)$, он разбивается на два полуинтервала: $[ind, ind + 2^{k-1})$ и $[ind + 2^{k-1}, ind + 2^k)$. И в принципе, в этой теме, как

[1] Вычисление НОДа двух чисел занимает неконстантное время, поэтому НОД на отрезке мы будем искать все таки не за $O(1)$, а за время работы одного вызова функции gcd

и в почти любой другой, лучше всего думать всегда про полуинтервалы, а не про отрезки, чтобы не запутаться.

Давайте приведем полный код построения разреженной таблицы за $O(n \log n)$:

```
vector<vector<int>> build_sparse_table(const vector<int>&
    arr) {
    int n = arr.size();
    int maxpow = ceil(log2(n + 1)); // 2^{maxpow} > n
    vector<vector<int>> sparse(maxpow, vector<int>(n, 0));
    sparse[0] = arr;
    for (int k = 0; k + 1 < maxpow; k++) {
        for (int ind = 0; ind + (1 << k) < n; ind++) {
            sparse[k + 1][ind] = min(sparse[k][ind],
                sparse[k][ind + (1 << k)]);
        }
    }
    return sparse;
}
```

Здесь для удобства выбран немного иной способ подсчета: мы пересчитываем не k-й слой через $k-1$-й, а $k+1$-й через k-й.

Обратите внимание, что для полной корректности выбранного нами подхода условие цикла по *ind* должно было выглядеть следующим образом:

`ind + (1 << (k + 1)) <= n`

То есть полуинтервал $[ind, ind + 2^{k+1})$ полностью лежит в границах массива. Однако мы ослабили это условие, посчитав некоторое количество лишних значений, однако мы все равно не используем такие значения при ответе на запрос, поэтому это никак не повлияет на работу нашей структуры. Но с другой стороны здесь нам не пришлось думать и вспоминать корректное условие. Все, о чем мы беспокоимся, — это то, чтобы индексы массива, через которые мы пересчитываем, были корректны. А так как мы обращаемся к индексу $ind + 2^k$, он должен быть меньше n.

maxpow можно выбрать любым способом, чтобы оно было строго больше $\log n$.

> **Замечание 4.2.1** Порядок циклов и размерностей в построении может очень сильно влиять на время исполнения программы из-за работы с кэшами. Предложенный вариант является оптимальным (примерно в 3-5 раз быстрее других вариантов).

4.3 Ответ на запрос

Для ответа на запрос нам надо найти ближайшую к длине отрезка степень двойки, а потом взять минимум из двух таких подотрезков. Не рекомендуется использовать для этого функцию логарифм, потому что она работает с вещественными числами, поэтому с ней можно легко ошибиться из-за точности. Можно заранее создать массив, в котором для каждого числа от 1 до n предпосчитать двоичный логарифм. Это делается следующим несложным кодом:

```
logs[1] = 0;
for (int i = 2; i <= n; i++) {
    logs[i] = logs[i >> 1] + 1;
}
```

Действительно, целая часть двоичного логарифма — это то, сколько раз число надо поделить нацело на два, чтобы оно стало единицей. Это можно посчитать как раз такой динамикой: если числу $\lfloor \frac{i}{2} \rfloor$ нужно `logs[i >> 1]` действий, то числу i нужно на одно действие больше.

Теперь можно и в построении *maxpow* определять как `logs[n] + 1`.

Когда мы посчитали логарифмы, мы готовы написать функцию поиска минимума на полуинтервале:

```
int find_min(int l, int r) { // [l, r)
    int power = logs[r - l];
    return min(sparse[power][l], sparse[power][r - (1 <<
        power)]);
}
```

Очевидно, запрос поиска минимума работает за $O(1)$.

Один из полуинтервалов, на которые мы разбиваем, должен начинаться в левой границе, а другой должен заканчиваться в правой, поэтому его левая граница, соответственно, должна быть на 2^{power} назад.

Обратите внимание, что благодаря тому, что мы все считаем на полуинтервалах, у нас во всей функции нет ни одной ± 1.

Кроме того, есть трюк, которым можно воспользоваться в C++, чтобы не предподсчитывать логарифмы. Конечно, функцию `log2` использовать не стоит, потому что она работает с числами с плавающей точкой, однако есть способ получить двоичный логарифм в целых числах. В этом нам поможет функция `__builtin_clz`. `builtin` означает «встроенная», а `clz` расшифровывается как «count leading zeros», то есть количество ведущих нулей. Заметим, что количество ведущих нулей как раз таки совпадает с индексом первой единицы (в числе идет k нулей, а после них стоит единица; это k-я позиция). А индекс старшей единицы — это как раз таки логарифм числа. Однако проблема в том, что в этом случае индекс считается с другой стороны: с самого старшего 31 бита, а нам нужен индекс с самого младшего бита, поэтому для получения целой части двоичного логарифма числа i, нужно воспользоваться следующей формулой:

`31 - __builtin_clz(i)`

> **Замечание 4.3.1** Есть альтернатива — функция `__lg(i)`, которая сразу возвращает целую часть логарифма, однако она есть не во всех компиляторах, поэтому будьте осторожны.

4.4 Применения

Применять разреженные таблицы можно и просто для поиска минимума на отрезке как более простую альтернативу дереву отрезков, так и в специальных случаях. К примеру, можно свести задачу о поиске наименьшего общего предка в дереве к задаче RMQ по высотам на эйлеровом обходе дерева.

4.5 Разреженные таблицы для поиска суммы на отрезке

Разреженные таблицы используют тот факт, что операция, которая считается на отрезке, идемпотентна, то есть если учесть один элемент два раза, ответ не поменяется. Однако такие операции как сумма не являются идемпотентными, поэтому из стандартных отрезков разреженных таблиц нельзя за $O(1)$ посчитать сумму на отрезке. Есть

альтернативный более сложный вариант построения разреженных таблиц (Disjoint Sparse Table), который разбивает любой отрезок на два непересекающихся подотрезка.

Однако сейчас давайте поймем, как искать сумму на отрезке при помощи стандартных разреженных таблиц за $O(\log n)$. Этот алгоритм будет похож на поиск предков в дереве при помощи бинарных подъемов. Давайте начнем с подотрезка, начинающегося в левой границе отрезка запроса, длины максимальной степени двойки и будем уменьшать степень, пока текущий подотрезок не попадет полностью внутрь отрезка запроса. После чего прибавим сумму на этом подотрезке к ответу и передвинем левую границу в конец этого подотрезка. Далее продолжим перебирать степени двойки по убыванию. Заметим, что если нам не подошла какая-то степень двойки, то далее она никогда нам не сможет подойти, потому что длина оставшегося отрезка запроса не увеличивается. Более того, набор взятых степеней двоек вовсе строго убывает. Действительно, если бы мы дважды взяли подотрезки длины 2^k, вместо этого можно было бы взять один подотрезок длины 2^{k+1}, но мы этого почему-то не сделали. Так что весь отрезок запроса разобьется на набор непересекающихся подотрезков различных длин. Всего таких подотрезков будет не больше логарифма, потому что всего разных степеней двоек есть логарифм.

> **Замечание 4.5.1** Если начинать поиск не с самой большой степени двойки, а с самой большой степени двойки, которая не больше длины отрезка (`power` из поиска минимума на отрезке), то асимптотика ответа на запрос будет $O(\log len)$, где len — длина отрезка запроса, что, конечно, в общем случае то же самое, что и $O(\log n)$.

```
1  long long find_sum(int l, int r) { // [l, r)
2      long long ans = 0;
3      for (int power = logs[r - l]; power >= 0; power--) {
4          if (l + (1 << power) <= r) {
5              ans += sparse[power][l];
6              l += (1 << power);
7          }
8      }
9      return ans;
10 }
```

4.6 Многомерные разреженные таблицы

Большим плюсом разреженных таблиц является простота их применения в многомерных случаях. Для примера рассмотрим двумерный случай, для бóльших размерностей все будет аналогично.

Если раньше отрезок длины n мы покрывали двумя подотрезками длины 2^k, где 2^k — ближайшая к n степень двойки, то теперь прямоугольник размера $n \times m$ мы накроем четырьмя подпрямоугольниками размера $2^k \times 2^l$ (ближайшие степени двойки к n и m соответственно), расположенными в углах прямоугольника запроса. То есть мы накрываем вертикальный отрезок длины n двумя подотрезками длины 2^k, накрываем горизонтальный отрезок длины m двумя подотрезками длины 2^l, и берем прямоугольники, полученные из всевозможных пар вертикальных и горизонтальных отрезков.

Построение двумерной разреженной таблицы можно устроить множеством разных способов. К примеру, можно сначала построить одномерные разреженные таблицы

по одной координате для каждой строки, а затем построить двумерную разреженную таблицу на этих одномерных.

Запрос нахождения минимума на прямоугольнике аналогичен одномерному случаю. В нем мы находим две координаты по одной оси, две координаты по другой и берем минимум среди подпрямоугольников, полученных всеми возможными их парами.

Очевидно, построение такой структуры будет занимать $O(n \cdot m \cdot \log n \cdot \log m)$ времени, а ответ на запрос все еще будет работать за $O(1)$.

С реализацией можно ознакомиться по ссылке.

4.7 Задачи для практики

- Задача на поиск максимума и индекса этого максимума на отрезке.
- Задачу E этого контеста сложно решить чем-то кроме разреженных таблиц.
- Задачи на Disjoint Sparse Table можно прорешать в специально подготовленном контесте на codeforces.

5. Дерево отрезков снизу

Дерево отрезков — крайне функциональная структура данных, с помощью которой можно решить огромное количество задач. Однако оно достаточно медленное: производятся рекурсивные вызовы и тому подобное. Иногда это приводит к тому, что решение может не проходить по времени. Кроме того, стандартное дерево отрезков — это не самый очевидный для написания алгоритм.

Чтобы решить эти проблемы часто прибегают к использованию дерева Фенвика. Оно очень простое в реализации, очень быстро работает, а также его легко обобщать для бо́льших размерностей. Однако оно ограничено тем, что в своей базовой вариации оно подходит только для обратимых операций, потому что считает функцию на отрезке через два префикса. То есть, к примеру, минимум на отрезке стандартным деревом Фенвика посчитать не получится. Кроме того, стандартное дерево Фенвика поддерживает изменение только в точке, но не на отрезке. В таких ситуациях вам может захотеться прибегнуть к использованию дерева отрезков снизу. Это нерекурсивная структура данных, которая одновременно работает так же быстро, как и дерево Фенвика, и при этом так же (или почти) функциональна как обычное дерево отрезков.

5.1 Изменение в точке, сумма на отрезке

Идея скрывается в названии. Если в обычном дереве отрезков мы начинаем с корня и идем вниз, разбивая отрезок запроса на логарифм подотрезков, то в дереве отрезков снизу мы начинаем с листьев и поднимаемся вверх.

Для начала давайте дополним длину массива до ближайшей степени двойки, чтобы он полностью лежал на одном уровне дерева. От этого асимптотика никак не поменяется.

Давайте сначала рассмотрим базовую задачу. Есть запросы двух видов: изменить значение в точке и найти сумму на отрезке.

Изменение в точке реализуется очень просто. Мы изменяем значение в листе, соответствующем этой точке, а затем постепенно идем до корня, обновляя значения в вершинах на пути через значения в детях.

Поиск суммы на отрезке тоже весьма прост. Вместо того, чтобы суммировать отрезок из k значений на текущем уровне, мы можем перейти на предыдущий уровень и там просуммировать лишь $\frac{k}{2}$ значений, ведь элементы более высокого уровня отвечают за сумму двух соседних элементов текущего уровня. Однако если левая граница отрезка является правым сыном, либо правая граница является левым сыном, то мы не можем перейти для них на предыдущий уровень, и их значения надо прибавить на текущем этапе. Так мы будем постепенно подниматься по дереву, делая на каждом уровне константное количество операций.

Асимптотика изменения равна $O(\log n)$, потому что состоит просто из прохода от листа к корню, а асимптотика нахождения суммы на полуинтервале $[l, r)$ равна $O(\log(r-l))$, что в общем случае, конечно, равно $O(\log n)$.

Построение дерева выглядит совсем просто. Мы записываем элементы изначального массива в листья подряд, а затем проходим по внутренним вершинам по убыванию, пересчитывая значение текущей вершины через детей.

```
1  const int N = (1 << 20); // N = 2^k, N >= n
2  long long tree[2 * N];
3
4  void build(const vector<int>& arr) {
5      for (size_t i = 0; i < arr.size(); i++) {
6          tree[N + i] = arr[i];
7      }
8      for (int i = N - 1; i > 0; i--) {
9          tree[i] = tree[i << 1] + tree[(i << 1) | 1];
10     }
11 }
12
13 void updatePoint(int pos, int newval) { // arr[pos] := newval
14     pos += N;
15     tree[pos] = newval;
16     pos >>= 1;
17     while (pos > 0) {
18         tree[pos] = tree[pos << 1] + tree[(pos << 1) | 1];
19         pos >>= 1;
20     }
21 }
22
23 long long find_sum(int l, int r) { // [l, r)
24     l += N;
25     r += N;
26     long long ans = 0;
27     while (l < r) {
28         if (l & 1) {
29             ans += tree[l++];
30         }
31         if (r & 1) {
32             ans += tree[--r];
33         }
34         l >>= 1;
35         r >>= 1;
```

```
36      }
37      return ans;
38  }
```

Мы заменили все операции на битовые для ускорения. Отец вершины `v` — это `v >> 1`, а сыновья — `v << 1` и `(v << 1) | 1`. Вершина является правым сыном, если ее номер нечетный, а левым, если четный (обратите внимание, что и для левой, и для правой границы условия в `find_sum` одинаковые, потому что правая граница берется не включительно).

Однако самое удивительное — это то, что на самом деле везде в этом коде вместо `N` можно написать `n`:

```
1   vector<long long> tree;
2   int n;
3
4   void build(const vector<int>& arr) {
5       n = arr.size();
6       tree.assign(2 * n, 0);
7       for (int i = 0; i < n; i++) {
8           tree[n + i] = arr[i];
9       }
10      for (int i = n - 1; i > 0; i--) {
11          tree[i] = tree[i << 1] + tree[(i << 1) | 1];
12      }
13  }
14
15  void update_point(int pos, int newval) { // arr[pos] := newval
16      pos += n;
17      tree[pos] = newval;
18      pos >>= 1;
19      while (pos > 0) {
20          tree[pos] = tree[pos << 1] + tree[(pos << 1) | 1];
21          pos >>= 1;
22      }
23  }
24
25  long long find_sum(int l, int r) { // [l, r)
26      l += n;
27      r += n;
28      long long ans = 0;
29      while (l < r) {
30          if (l & 1) {
31              ans += tree[l++];
32          }
33          if (r & 1) {
34              ans += tree[--r];
35          }
36          l >>= 1;
37          r >>= 1;
38      }
```

```
39     return ans;
40 }
```

Замечание 5.1.1 Если вам нужно работать с некоммутативной операцией, то нужно комбинировать ответ правильно. Нужно отдельно хранить левый префикс ответа и правый суффикс, а в конце их объединять.

Таким образом, наша структура будет занимать ровно $2n$ памяти в отличие от обычного дерева отрезков, которому в зависимости от реализации бывает нужно от $4n$ до $8n$ памяти.

Конечно, для операции суммы можно использовать и дерево Фенвика, но так как у нас есть вся мощь дерева отрезков, мы можем считать практически любую другую функцию на отрезке. К примеру, минимум, с которым стандартное дерево Фенвика не справится.

5.2 Изменение на отрезке, значение в точке

Как обычно, структуру, которая умеет изменять в точке и находить значение функции на отрезке, можно переделать в структуру, которая умеет изменять целый отрезок и находить значение в точке. Для этого нам надо просто поменять местами функцию изменения с функцией нахождения значения. Функция изменения теперь разобьет отрезок запроса на логарифм вершин дерева отрезков, в каждой из которых она оставит изменение, которое нужно сделать со всем поддеревом. Функция нахождения значения в точке в свою очередь пройдется по пути до корня и посчитает ответ.

Замечание 5.2.1 Обратите внимание, что в этом случае нам необходимо, чтобы операции изменения на отрезке были коммутативны, то есть порядок их применения был неважен. К примеру, для операции прибавления на отрезке это верно, а для операции присвоения на отрезке это неверно. Дело в том, что мы оставляем изменения в вершинах дерева, а когда пытаемся найти значение, идем по порядку по вершинам, но изменения в них могут быть перемешаны во времени. Конкретно для присвоения на отрезке можно кроме самого присвоения хранить еще и момент времени, когда оно было сделано, а потом выбрать максимум таких моментов, однако в общем случае, возможно, что все операции придется применить в правильном порядке, и тогда нам придется их отсортировать, поэтому запрос нахождения значения в точке будет работать за $O(\log n \cdot \log\log n)$. Для некоммутативных операций нам подойдет дерево отрезков с ленивыми обновлениями, которое мы обсудим позже.

```
1  vector<long long> tree;
2  int n;
3
4  void build(const vector<int>& arr) {
5      n = arr.size();
6      tree.assign(2 * n, 0);
7      for (int i = 0; i < n; i++) {
8          tree[n + i] = arr[i];
9      }
10     // tree[0..n-1] are zeros because there's nothing to add
           on a segment
11 }
```

```
12
13 long long find_value(int pos) {
14     pos += n;
15     long long ans = 0;
16     while (pos > 0) {
17         ans += tree[pos];
18         pos >>= 1;
19     }
20     return ans;
21 }
22
23 void segment_update(int l, int r, int addval) { // [l, r)
24     l += n;
25     r += n;
26     while (l < r) {
27         if (l & 1) {
28             tree[l++] += addval;
29         }
30         if (r & 1) {
31             tree[--r] += addval;
32         }
33         l >>= 1;
34         r >>= 1;
35     }
36 }
```

Обратите внимание, что в 11 строке мы не пересчитываем значение в вершине через детей, потому что теперь мы в вершине храним не сумму на отрезке, а значение, которое нужно прибавить ко всем числам на отрезке.

5.3 Двумерные запросы

Когда дерево отрезков двумерное, оно становится совсем медленным, и нам нужно делать сложную схему из рекурсивного запуска запроса к одномерному дереву отрезков из двумерного. В случае нерекурсивной реализации дерева отрезков снизу все практически так же просто, как с деревом Фенвика. Необходимо лишь написать два вложенных цикла вместо одного:

```
1  long long find_sum(int lx, int rx, int ly, int ry) { // [lx,
       rx) * [ly, ry)
2      lx += n;
3      rx += n;
4
5      long long ans = 0;
6      while (lx < rx) {
7          int curly = ly + m;
8          int curry = ry + m;
9          while (curly < curry) {
10             if (curly & 1) {
11                 if (lx & 1) {
12                     ans += tree[lx][curly];
```

```
13                  }
14                  if (rx & 1) {
15                      ans += tree[rx - 1][curly];
16                  }
17              }
18              if (curry & 1) {
19                  if (lx & 1) {
20                      ans += tree[lx][curry - 1];
21                  }
22                  if (rx & 1) {
23                      ans += tree[rx - 1][curry - 1];
24                  }
25              }
26              curly = (curly + 1) >> 1;
27              curry >>= 1;
28          }
29          lx = (lx + 1) >> 1;
30          rx >>= 1;
31      }
32      return ans;
33  }
```

Нужно сдвинуть границу в том случае, если она нечетна и по x координате, и по y. С полной реализацией можно ознакомиться по ссылке.

5.4 Ленивые обновления

Чаще всего так получается, что обычно хватает обновления в точке. Либо же если все-таки нужно обновление на отрезке, то ограничения не такие жестокие, и можно использовать обычное дерево отрезков. Однако давайте все-таки поймем, как с помощью дерева отрезков снизу можно поддерживать изменения на отрезке и запрос поиска на отрезке. В этом нам помогут (как и в обычном дереве отрезков) ленивые обновления.

Мы будем поддерживать все то же самое, что и в обычном дереве отрезков, но нужно просто перевести это на язык дерева отрезков снизу.

Когда мы делаем запрос поиска на отрезке, нам нужно протолкнуть отложенные изменения из всех предков вершин, которые мы посетим. Все эти вершины — это просто предки листьев, соответствующих концам отрезка запроса. Однако проталкивать изменения надо сверху вниз, чтобы они комбинировались друг с другом. Как же это сделать в дереве отрезков снизу? Мы знаем, что предок вершины v — это $\left\lfloor \frac{v}{2} \right\rfloor$, тогда k-й предок — это $\left\lfloor \frac{v}{2^k} \right\rfloor$. Поэтому мы можем просто перебрать степень двойки по убыванию и проталкивать изменения сверху вниз.

Запрос изменения на отрезке будет выглядеть так же, как запрос поиска суммы на отрезке, только в конце после того как мы изменили какие-то значения, нужно пересчитать значения предков измененных вершин. Но это как и в прошлом случае просто предки листьев, отвечающих за концы отрезка. Однако в этот раз перебирать их надо уже как обычно — снизу вверх.

Также стоит обратить внимание на то, что для вершин с номерами $[n, 2n)$ не нужно хранить ленивые обновления, потому что они являются листьями, поэтому в случае жестких ограничений по памяти можно достичь использования $3n$ ячеек.

Рассмотрим код на примере задачи прибавления на отрезке и поиска максимума на отрезке:

```
const long long INF = 1e18;
vector<long long> tree;
vector<long long> push;
int n;
int logn;

void build(const vector<int>& arr) {
    n = arr.size();
    logn = 32 - __builtin_clz(2 * n);
    tree.assign(2 * n, 0);
    push.assign(n, 0);
    for (int i = 0; i < n; i++) {
        tree[n + i] = arr[i];
    }
    for (int i = n - 1; i > 0; i--) {
        tree[i] = max(tree[i << 1], tree[(i << 1) | 1]);
    }
}

void update_vertex(int v, long long val) {
    tree[v] += val;
    if (v < n) {
        push[v] += val;
    }
}

void update_ancestors(int v) {
    v >>= 1;
    while (v > 0) {
        tree[v] = max(tree[v << 1], tree[(v << 1) | 1]) +
           push[v];
        v >>= 1;
    }
}

void do_push(int leaf) {
    for (int k = logn; k > 0; k--) {
        int v = (leaf >> k);
        update_vertex(v << 1, push[v]);
        update_vertex((v << 1) | 1, push[v]);
        push[v] = 0;
    }
}

void update_segment(int l, int r, int val) { // [l, r) += val
    l += n;
    r += n;
```

```
47     int ql = l, qr = r;
48     while (l < r) {
49         if (l & 1) {
50             update_vertex(l++, val);
51         }
52         if (r & 1) {
53             update_vertex(--r, val);
54         }
55         l >>= 1;
56         r >>= 1;
57     }
58     update_ancestors(ql);
59     update_ancestors(qr - 1);
60 }
61 
62 long long find_max(int l, int r) { // [l, r)
63     l += n;
64     r += n;
65     do_push(l);
66     do_push(r - 1);
67     long long ans = -INF;
68     while (l < r) {
69         if (l & 1) {
70             ans = max(ans, tree[l++]);
71         }
72         if (r & 1) {
73             ans = max(ans, tree[--r]);
74         }
75         l >>= 1;
76         r >>= 1;
77     }
78     return ans;
79 }
```

В более сложных случаях (к примеру, поиска суммы на отрезке) для пересчета значения в функции `apply` может понадобиться длина отрезка текущей вершины. И в случае, когда мы идем снизу вверх, и в случае, когда мы идем сверху вниз, легко поддерживать эту величину.

5.5 Задачи для практики

Можно потренироваться в первую очередь на любых задачах на дерево отрезков, коих существует бесчисленное количество.

К примеру, можно воспользоваться следующими задачами:

- Задача на поиск максимума на отрезке.
- Задача на прибавление на отрезке, присвоение на отрезке и поиск суммы на отрезке.
- Задача на прибавление арифметической прогрессии на отрезке.
- Чуть более продвинутая задача на дерево отрезков.
- ОСТОРОЖНО! СПОЙЛЕРЫ К РОИ!

Эту задачу можно очень просто сдать на высокий балл при помощи дерева отрезков снизу. Такое решение выигрывает у разреженных таблиц не смотря на то, что они отвечают на запрос за $O(1)$.

6. Segment Tree Beats

Segment Tree Beats (STB) — это структура данных[1], которая была разработана Ruyi jiry_2 Ji в 2016 году. Это очень мощный инструмент, идея которого состоит в том, что мы ослабляем условия выхода из рекурсии в дереве отрезков, в результате чего кажется, что алгоритм начинает работать за квадратичное время, но при помощи амортизационного анализа можно доказать, что на самом деле время работы сильно меньше ($O(n \log n)$, $O(n \log^2 n)$ и т.д.). Также эта структура данных позволяет работать с «исторической информацией» массива. Частным случаем Segment Tree Beats является структура Ji Driver Segment Tree, которая позволяет поддерживать операции вида «заменить все числа на отрезке массива A на $\max(A_i, x)$», а также узнавать сумму на отрезке.

На английском языке на эту тему есть по большому счету только одна статья. На русском же языке, насколько мне известно, материалов на эту тему нет в принципе. Статья, которую вы сейчас читаете, не только полностью покрывает англоязычный текст, но и затрагивает большое количество тем, которые в ней не упоминались, поэтому, пожалуй, является самым полным материалом про Segment Tree Beats не на китайском языке на данный момент. Я попытался собрать все возможные идеи, которые есть на эту тему, а также дополнить несколькими своими.

На данный момент в русском языке нет какой-либо используемой альтернативы английскому названию, но если вы предпочитаете локализацию, то есть вариант «Анимешное Дерево Отрезков»[2].

В этой статье мы часто будем говорить про асимптотику. Всегда подразумевается, что n — это размер массива, а q — суммарное количество запросов.

[1] Назвать это структурой данных можно весьма условно. Это скорее набор идей, которые наслаиваются на дерево отрезков.

[2] codeforces.com/blog/entry/90460?locale=ru#comment-789207

6.1 Общая идея

Давайте посмотрим, как выглядит стандартная функция изменения в дереве отрезков с массовым обновлением и проталкиванием:

```
void update(int node, int l, int r, int ql, int qr, int newval) {
    if (qr <= l || r <= ql) { // node is outside of the segment
        return;
    }
    if (ql <= l && r <= qr) { // node is inside the segment
        update_node(node, newval);
        set_push(node, newval);
        return;
    }
    // node intersects the segment
    push_down(node);
    int mid = (r + l) / 2;
    update(2 * node, l, mid, ql, qr, newval);
    update(2 * node + 1, mid, r, ql, qr, newval);
    pull_up(node);
}
```

Пускай мы находимся в вершине `node`, которая отвечает за полуинтервал $[l,r)$ массива, и нас попросили обновить значения массива на полуинтервале $[ql,qr)$ значением `newval`.

Первое условие (`break_condition`) проверяет, что если отрезок, за который отвечает вершина `node`, не пересекается с отрезком, на котором мы делаем обновление, то в текущем поддереве ничего менять не надо, и можно просто вернуться назад.

Второе условие (`tag_condition`) проверяет, что если отрезок, за который отвечает вершина `node`, лежит полностью внутри отрезка, который мы обновляем, то мы обновим значение прямо здесь, а также сохраним `push`, который в будущем будем проталкивать в детей. После чего мы опять же завершаемся и возвращаемся назад.

Если же ни первое, ни второе условие не выполнились, то это значит, что отрезки запроса и текущей вершины пересекаются, но при этом текущая вершина не лежит полностью внутри запроса. В таком случае мы рекурсивно запускаемся из детей, не забыв предварительно протолкнуть информацию о старых обновлениях, а после завершения работы в детях восстанавливаем значение в текущей вершине через значения детей.

Этот код будет работать за $O(\log n)$, потому что на каждом уровне дерева отрезков не больше, чем две вершины могут пересекаться с отрезком запроса, но при этом не лежать в нем полностью, поэтому только из этих двух вершин мы рекурсивно запустимся на следующий уровень, а значит, на каждом уровне дерева мы посетим не более четырех вершин.

Segment Tree Beats основан на следующей идее: пускай запросы изменения таковы, что мы не всегда можем пересчитать значение на отрезке при условии выполнения `tag_condition`. Тогда давайте усилим условие `break_condition` и ослабим условие `tag_condition`, чтобы теперь мы могли уже пересчитать значение в вершине, не запускаясь рекурсивно, но при этом асимптотика не стала квадратичной.

То есть, теперь функция изменения будет выглядеть следующим образом:

```
1  void update(int node, int l, int r, int ql, int qr, int
      newval) {
2      if (break_condition(node, ql, qr, newVal)) {
3          return;
4      }
5      if (tag_condition(node, ql, qr, newval)) {
6          update_node(node, newval);
7          set_push(node, newval);
8          return;
9      }
10     push_down(node);
11     int mid = (r + l) / 2;
12     update(2 * node, l, mid, ql, qr, newval);
13     update(2 * node + 1, mid, r, ql, qr, newval);
14     pull_up(node);
15 }
```

Иными словами, все, что нам нужно сделать — это придумать наиболее сильное условие `break_condition`, при котором в текущем поддереве запрос изменения точно ничего не изменит, а также наиболее сильное условие `tag_condition`, при котором можно будет обновлять значение в текущей вершине, не запускаясь рекурсивно из детей.

При этом заметьте, что `break_condition` и `tag_condition` из обычного дерева отрезков никуда не деваются. Скорее всего, если отрезки запроса и текущей вершины не пересекаются, то в этой вершине точно ничего не надо менять. С другой стороны, если текущая вершина не лежит полностью внутри отрезка запроса, то вряд ли можно пересчитать значение в ней, не запустившись рекурсивно в детей, так что эти условия будут выглядеть примерно следующим образом:

`break_condition = qr <= l || r <= ql || ???`

`tag_condition = ql <= l && r <= qr && ???`

В этой статье мы будем пытаться придумать, чем нужно заменить каждый из `???` в разных задачах.

Задачи в основном будут описываться запросами, которые в них нужно выполнять. К примеру, запрос `+=` означает, что необходимо уметь прибавлять какое-то значение на отрезке. Запрос `=` означает, что необходимо уметь присваивать какое-то значение на отрезке. Эти запросы являются весьма стандартными для дерева отрезков. Однако кроме них будут рассмотрены и более сложные: `max=`, `min=`, `%=`, `/=` и так далее. Они означают, что в результате запроса нужно заменить все элементы на отрезке на результат выполнения соответствующей функции от текущего значения и `newval`. К примеру, операция `max=` заменят все элементы на отрезке массива A по правилу $A_i \to \max(A_i, newval)$.

Кроме того, не менее важно, какие операции типа `get` есть в задаче. К примеру, мы будем рассматривать следующие операции: $\sum$ — сумма на отрезке, $\max$ — максимум на отрезке, $\min$ — минимум на отрезке, gcd — НОД на отрезке и т.д.

> **Замечание 6.1.1** Обратите внимание, что запрос типа `get`, то есть запрос получения какой-либо функции на отрезке, не меняется, потому что мы все еще поддерживаем корректную информацию о подотрезке вершины, когда мы в нее спустились от корня.

6.2 %=, = в точке, $\sum$

6.2.1 Формулировка

В этой задаче у нас есть массив неотрицательных целых чисел A $(0 \leqslant A_i < C)$, а также имеются запросы трех типов:

1. Даны ql, qr, x $(1 \leqslant x < C)$. Нужно заменить все элементы массива A на полуинтервале $[ql, qr)$ на $A_i \bmod x$.
2. Даны qi, y $(0 \leqslant y < C)$. Нужно заменить элемент массива A на позиции qi на y.
3. Даны ql, qr. Необходимо вернуть сумму элементов массива A на полуинтервале $[ql, qr)$.

Эту задачу можно найти здесь.

Также можно без проблем поддерживать и другие `get`-запросы, такие как max или min.

6.2.2 Решение

Вторую и третью операции мы будем выполнять как обычно. Осталось понять, в какой момент мы можем остановиться в первом запросе, чтобы обновить значение в текущей вершине, а также в будущем иметь возможность эффективно проталкивать это изменение в детей.

Давайте подумаем, каким должно быть `break_condition`? При каком условии ни одно число в данном поддереве не поменяется? В том случае, если все числа на подотрезке текущей вершины меньше, чем x. Иными словами, если максимум на этом отрезке меньше x. Поэтому `break_condition` в данном случае будет выглядеть следующим образом: `qr <= l || r <= ql || maxvalue[node] < x`.

Теперь надо придумать такое условие `tag_condition`, при котором нам не придется идти в детей. Здесь уже может быть несколько вариантов. К примеру, если целые части от деления всех чисел на отрезке на x совпадают. Однако нам хватит более простого условия: если все числа на отрезке равны, то есть, иными словами, максимум на отрезке равен минимуму. В этом случае все числа на отрезке равны `maxvalue[node]`, поэтому операция %= на этом отрезке — это то же самое, что присвоить на этом отрезке всем элементам `maxvalue[node]` $\bmod x$. Это мы можем сделать так же, как и с запросом второго типа, не заходя в детей.

6.2.3 Доказательство

Осталось понять, почему при таком ослаблении `tag_condition` асимптотика остается приемлимой. На самом деле асимптотика этого решения — $O((n+q)\log n \log C)$.

Введем потенциал вершины дерева отрезков $\varphi(node)$, равный $\sum_{l \leqslant i < r} \log(A_i + 1)$, где l, r — границы полуинтервала исходного массива, за который отвечает данная вершина. Прибавление единицы, конечно, ни на что особо влиять не будет, но оно необходимо, потому что A_i могут быть равны нулю. Теперь введем потенциал Φ, который будет равен сумме потенциалов всех вершин дерева. В любой момент времени потенциал можно оценить следующим образом: $0 \leqslant \Phi \leqslant O(n \log n \log C)$, потому что каждый элемент массива лежит в поддереве у $\log n$ вершин дерева отрезков и дает вклад $O(\log C)$.

И пускай суммарно за все время этот потенциал увеличился на Φ_+. Давайте разобьем вершины, которые посетит запрос изменения, на три вида: **обычные**, **дополнительные** и **тупиковые**. Обычные вершины — это те вершины, которые посетило бы стандартное дерево отрезков, тупиковые вершины — это те необычные вершины, в которых выполнилось одно из условий `break_condition` или `tag_condition`, а дополнительные вершины — это все остальные вершины, которые посетил запрос, то есть те не обычные вершины, из которых мы вызвались рекурсивно. Тогда заметим,

что некоторые верхние вершины дерева будут обычными, потом под обычными будет какое-то количество дополнительных, и из дополнительных иногда будут торчать тупиковые. Из тупиковых уже рекурсивных вызовов нет. При этом заметим, что отец любой тупиковой вершины — это либо обычная вершина, либо дополнительная. Кроме того, у каждой вершины максимум 2 сына, так что тупиковых вершин максимум в два раза больше, чем обычных и дополнительных, поэтому их посещение не влияет на асимптотику, и можно следить только за дополнительными вершинами. Это можно было понять немного иначе: если мы делаем рекурсивный вызов и сразу завершаемся, то в каком-то смысле можно считать, что этот рекурсивный вызов не был сделан вовсе.

Теперь, когда мы оставили только обычные и дополнительные вершины, докажем, что при посещении дополнительной вершины потенциал в этой вершине (а следовательно и суммарный потенциал Φ) уменьшается хотя бы на 1, тогда суммарное количество посещенных дополнительных вершин можно оценить как $O(n \log n \log C) + \Phi_+$, а обычных вершин мы на каждом запросе посещаем $O(\log n)$ штук. При этом в каждой вершине дерева мы делаем константное количество операций, так что асимптотика алгоритма будет равна $O(n \log n \log C + \Phi_+ + q \log n)$. Остается только показать, что $\Phi_+ \leqslant O(q \log n \log C)$, а также то, что потенциал уменьшается при посещении дополнительной вершины.

Данное рассуждение может показаться сложным и запутанным, но на самом деле в будущем все рассуждения об амортизированном времени работы будут очень похожи на это. Мы вводим какой-то потенциал. Понимаем, что он всегда находится в пределах от 0 до $\max \Phi$, смотрим, на сколько он может увеличиваться, а также, на сколько он уменьшается при посещении дополнительных вершин. Из этого делается вывод о времени работы алгоритма.

> **Замечание 6.2.1** Это не совсем обычный способ измерения амортизированного времени работы. Обычно вводят $a_i = t_i - \Delta\Phi_i$, после чего получается, что время работы можно оценить как $\Delta\Phi + \sum a_i$. Однако в данном контексте такие рассуждения, пожалуй, менее удобны для понимания. Мы воспринимаем потенциал как кучку камней, которая может иметь ограниченный размер, за все время в нее положат какое-то конкретное ограниченное количество камней, а за каждую необычную операцию мы будем забирать из этой кучки камень.

Итак, давайте поймем, чему равен Φ_+. Операция первого типа может только уменьшать числа в массиве, а операция третьего типа вовсе не меняет элементов массива. Поэтому увеличения потенциала могут происходить только во время операций второго типа. Мы изменили один элемент массива. Он был $\geqslant 0$, а стал $< C$, поэтому потенциал мог увеличиться максимум на $\log((C-1)+1) - \log(0+1) = \log C$ для каждой вершины, в поддереве которой есть этот элемент, а таких вершин $O(\log n)$. Всего запросов было q, поэтому суммарно потенциал увеличится максимум на $O(q \log n \log C)$. Что и требовалась показать.

Теперь покажем, почему при посещении дополнительной вершины ее потенциал уменьшается как минимум на 1. Если мы посещаем дополнительную вершину, это значит, что ко всем элементам на ее подотрезке нужно применить операцию %=, и при этом на этом отрезке есть хотя бы одно число, которое не меньше x. Воспользуемся следующим известным фактом:

> **Теорема 6.2.2** Если $k \geqslant x$, то $k \bmod x \leqslant \frac{k-1}{2}$.

Доказательство. Во-первых, заметим, что условие $l \leqslant \frac{k-1}{2}$ для целых l и k равносильно

тому, что $l < \frac{k}{2}$.

Во-вторых, разберем два случая:

1. $k \geqslant 2x$. В этом случае $k \bmod x < x \leqslant \frac{k}{2}$. Первое неравенство верно просто потому, что остаток от деления всегда меньше модуля, а второе верно из-за того, что $k \geqslant 2x$.
2. $k < 2x$. В этом случае $k \bmod x = k - x < \frac{k}{2}$ в силу того, что $k < 2x$.

■

То есть, если мы посетили дополнительную вершину, то какое-то число на этом отрезке уменьшится больше, чем в два раза. Тогда раньше это число давало вклад $\log(k+1)$ в потенциал, а теперь $\leqslant \log(\frac{k-1}{2}+1) = \log(\frac{k+1}{2}) = \log(k+1) - 1$. Таким образом, мы доказали, что потенциал этой вершины уменьшился хотя бы на один. Что и требовалось.

> **Замечание 6.2.3** В одной из последующих секций мы докажем, что абсолютно такое же решение будет работать за такую же асимптотику даже если присвоение происходит на отрезке.

6.3 `min=`, $\sum$, max (Ji Driver Segment Tree)

6.3.1 Формулировка

В этой задаче у нас есть массив целых чисел A, а также имеются запросы трех типов:

1. Даны ql, qr, x. Нужно заменить все элементы массива A на полуинтервале $[ql, qr)$ на $\min(A_i, x)$.
2. Даны ql, qr. Необходимо вернуть сумму элементов массива A на полуинтервале $[ql, qr)$.
3. Даны ql, qr. Необходимо вернуть максимум в массиве A на полуинтервале $[ql, qr)$.

Эта задача доступна здесь или здесь. Это китайские сайты, поэтому там не так просто зарегистрироваться.

Также есть усложненная версия этой задачи «Нагайна».

6.3.2 Решение

Это самая стандартная задача на Segment Tree Beats. Собственно, статья на английском появилась после того, как участники из Китая массово решали задачу «Нагайна» при помощи Ji Driver Segment Tree, хотя изначально эта задача была на корневую декомпозицию.

В этой задаче мы будем хранить в каждой вершине следующие значения: `sum` — сумма на отрезке; `max` — максимум на отрезке; `cnt_max` — количество элементов на этом отрезке, которые равны максимуму; `second_max` — наибольший элемент на отрезке, который строго меньше, чем `max`. Обратите внимание на то, что это строгий второй максимум, то есть если на отрезке есть числа $0, 1, 2, 3, 3$, то `second_max` $= 2$.

Каким должно быть `break_condition`? В каком случае операция `min=` не поменяет ничего на текущем отрезке? В том случае, если все элементы на этом отрезке уже не больше, чем x, и ничего менять не надо. То есть, другими словами, если максимум не больше x:

```
break_condition = qr <= l || r <= ql || max <= x
```

А каким же должно быть `tag_condition`? В каком случае мы можем быстро обновить значения в текущей вершине? В том случае, если меняются только максимумы, то есть `second_max < x`.

```
tag_condition = ql <= l && r <= qr && second_max < x
```

В таком случае мы знаем, что `cnt_max` максимумов заменятся на `x`, и в этом случае легко можно пересчитать все значения в вершине, и вниз мы будем проталкивать как раз эту информацию: с каким числом нужно произвести операцию `min=`. Если нужно протолкнуть две операции `min=`, то достаточно проталкивать всего одну такую операцию с меньшим из параметров.

На самом деле можно заметить, что проталкиваемое значение можно не хранить вовсе, потому что оно всегда совпадает с `max` в этой вершине (либо проталкивать ничего не надо, но если мы протолкнем максимум, ничего не изменится). То есть мы в каком-то смысле проталкиваем в детей наш максимум на отрезке. Так писать код становится сильно приятнее.

6.3.3 Доказательство

Докажем, что это решение работает за $O((n+q)\log n)$.

Опять же воспользуемся методом потенциалов. Определим потенциал вершины $\varphi(node)$ как количество различных чисел на отрезке, за который отвечает эта вершина. Общий потенциал Φ определяется опять же как сумма потенциалов по всем вершинам дерева.

Количество различных чисел на отрезке не больше, чем длина этого отрезка, так что сумма потенциалов на одном уровне дерева отрезков не больше n. Таким образом, в любой момент времени потенциал ограничен следующим образом: $0 \leqslant \Phi \leqslant O(n \log n)$.

Чему равно Φ_+? Во время `get`-запросов элементы массива не меняются, поэтому в них потенциал не увеличивается. В запросах первого типа потенциал может измениться только для обычных и дополнительных вершин, потому что для непосещенных вершин значения на отрезке не поменяются, а для тупиковых никакие значения не склеятся. При этом для дополнительных вершин количество различных чисел только уменьшается за счет того, что числа склеиваются, но увеличиться точно не может. Поэтому потенциал увеличивается только для обычных вершин, при этом единственное новое значение, которое может появиться, — это `x`, потому что каждое число либо не меняется, либо заменяется на `x`. Таким образом, потенциал мог увеличиться только у $O(\log n)$ обычных вершин, и для каждой такой вершины он увеличился максимум на 1. Поэтому за все время $\Phi_+ \leqslant O(q \log n)$.

Теперь поймем, что посещение дополнительных вершин уменьшает потенциал этой вершины хотя бы на 1. Это следует из того, что для дополнительных вершин $x \leqslant$ `second_max`, поэтому после выполнения этого запроса и `max`, и `second_max` заменятся на x, то есть два максимума как бы склеятся между собой, так что количество различных чисел в этой вершине точно уменьшится хотя бы на 1.

Таким образом, асимптотика алгоритма получается равной $O(n\log n + q\log n + q\log n) = O((n+q)\log n)$ (первое слагаемое от глобального изменения потенциала от начала до конца, второе слагаемое от Φ_+, а третье слагаемое от посещения обычных вершин). Что и требовалось доказать.

> **Замечание 6.3.1** По аналогии с этой задачей можно поддерживать также операцию `max=` и даже их совмещение. Достаточно хранить 2 максимума, 2 минимума и их количества. Оценка времени работы от этого не изменится.
>
> Кроме того, можно добавить операцию присвоения на отрезке, потому что она так же, как и `min=` не сильно увеличивает потенциал.

6.4 min=, max=, +=, ∑, max, min

6.4.1 Формулировка

В этой задаче у нас есть массив целых чисел A, а также имеются запросы шести типов:

1. Даны ql, qr, x. Нужно заменить все элементы массива A на полуинтервале $[ql, qr)$ на $\min(A_i, x)$.
2. Даны ql, qr, y. Нужно заменить все элементы массива A на полуинтервале $[ql, qr)$ на $\max(A_i, y)$.
3. Даны ql, qr, z. Нужно заменить все элементы массива A на полуинтервале $[ql, qr)$ на z.
4. Даны ql, qr, t. Нужно прибавить ко всем элементам массива A на полуинтервале $[ql, qr)$ число t.
5. Даны ql, qr. Необходимо вернуть сумму элементов массива A на полуинтервале $[ql, qr)$.
6. Даны ql, qr. Необходимо вернуть максимум элементов массива A на полуинтервале $[ql, qr)$.
7. Даны ql, qr. Необходимо вернуть минимум элементов массива A на полуинтервале $[ql, qr)$.

6.4.2 Решение

Решение никак не меняется. Оно такое же, как и раньше (только надо добавить стандартную операцию +=), однако теперь старое доказательство перестает работать. Нам надо будет придумать новое.

6.4.3 Доказательство

Почему же старое доказательство не работает?

По факту, новая операция здесь всего одна: +=, потому что по замечанию 6.3.1 остальные запросы просто встраиваются в Ji Driver Segment Tree. Однако с операцией += потенциал из прошлой задачи не пройдет. Давайте посмотрим на пример массива:

$$1, 2, 3, 4, \dots, \frac{n}{2} - 1, \frac{n}{2}, 1, 2, 3, 4, \dots, \frac{n}{2} - 1, \frac{n}{2}$$

Корень дерева отвечает за весь массив, поэтому его потенциал равен $\frac{n}{2}$. Однако если мы прибавим $\frac{n}{2}$ ко второй половине массива, то массив станет выглядеть так:

$$1, 2, 3, 4, \dots, n - 1, n$$

И в этом случае потенциал корня стал равен n, а потенциалы других вершин не изменились. То есть, мы за одну операцию увеличили потенциал на $\frac{n}{2}$. Это слишком много. Нам нужен другой потенциал.

Доказывать мы, однако, будем уже не $O((n+q)\log n)$, а $O(n \log n + q \log^2 n)$. Кроме того, вроде бы теста, на котором это работало бы за $\log^2 n$ на запрос неизвестно, так что, возможно, это настоящий $\log n$, но этим мы, конечно, пользоваться не будем.

В этот раз потенциалом дерева Φ_1 будет количество его вершин, для которых максимумы в левом и правом поддеревьях не совпадают. Скажем, что такие вершины являются помеченными. Аналогично, Φ_2 — это количество вершин, для которых минимумы в левом и правом поддеревьях не совпадают. Тогда очевидно, что в любой момент времени $0 \leqslant \Phi_1, \Phi_2 \leqslant O(n)$. В каких случаях потенциал Φ_1 мог увеличиваться? Если для какой-то вершины раньше максимумы в детях совпадали, а потом стали различаться. Это могло произойти только в том случае, если какие-то элементы на отрезке данной вершины поменялись, но не все. То есть текущая вершина пересекается с запросом, но не лежит в нем полностью. Это обязательно обычная вершина, таких

вершин $O(\log n)$ штук, так что за один запрос Φ_1 могло увеличиться максимум на $O(\log n)$. Аналогично для Φ_2. Так что $\Phi_{1_+}, \Phi_{2_+} \leqslant O(q\log n)$.

Теперь поймем, как уменьшаются потенциалы при операциях `min=` и `max=`. Докажем, что если мы посетили m дополнительных вершин в операции `min=`, то Φ_1 уменьшится хотя бы на $\frac{m}{\log n}$ (иными словами, если потенциал уменьшится на k, то мы посетим не больше $k\log n$ дополнительных вершин). Аналогично, для операции `max=`, уменьшаться будет Φ_2. Тогда итоговая асимптотика будет равна $O(n\log n + q\log^2 n)$.

Почему потенциал Φ_1 будет уменьшаться? Давайте докажем следующее утверждение:

Теорема 6.4.1 В поддереве любой дополнительной вершины v есть помеченная вершина u, которая после применения операции перестанет быть помеченной.

Доказательство. Если вершина v дополнительная, то для этой вершины `max[v]` $> x$ и `second_max[v]` $\geqslant x$, потому что не выполнились `break_condition` и `tag_condition`. Тогда докажем, что в поддереве вершины v есть такая вершина u, что для нее максимум в одном из детей равен `max[v]`, а в другом `second_max[v]`. И тогда сейчас эти числа различаются, а после применения операции оба будут равны x, поэтому из вершины u пропадет пометка.

Почему же такая вершина u существует? Посмотрим на самую глубокую вершину t в поддереве v, для которой `max[t] = max[v]` и `second_max[t] = second_max[v]`. Такая вершина точно есть, потому что как минимум подходит сама вершина v. Для вершины t это условие выполнено, а для ее детей — нет, потому что вершина t — это самая глубокая такая вершина. Тогда заметим, что вершина t как раз таки подходит на роль вершины u. Если максимум в поддереве t равен `max[v]`, то и у одного из сыновей максимум равен тому же самому числу. При этом второй максимум в этом сыне не равен `second_max[v]`, так как тогда этот сын был бы более глубокой подходящей вершиной. То есть в этом сыне нет значений, равных `second_max[v]`. Тогда все эти значения находятся в другом сыне. При этом в другом сыне не может быть значений, равных `max[v]`, потому что тогда этот другой сын был бы более глубокой вершиной, чем t, которая нам подходит. Так что максимум в другом сыне — это `second_max[v]`. Поэтому как раз таки вершина t подходит на роль вершины u, и в ней была метка, а после применения операции эта метка пропадет. ∎

Пускай после применения операции пропало k меток. Тогда все посещенные дополнительные вершины — это предки этих k вершин, потому что по теореме у любой дополнительной вершины есть потомок, в котором пропала метка. У каждой из этих k вершин есть $\log n$ предков, так что суммарно у них не более $k\log n$ предков, значит, мы посетим не больше, чем столько вершин. Что и требовалось доказать.

6.5 min=, +=, gcd

6.5.1 Формулировка

В этой задаче у нас есть массив неотрицательных целых чисел A $(0 \leqslant A_i < C)$, а также имеются запросы трех типов:

1. Даны ql, qr, x $(0 \leqslant x)$. Нужно заменить все элементы массива A на полуинтервале $[ql, qr)$ на $\min(A_i, x)$.
2. Даны ql, qr, y $(0 \leqslant y < C)$. Нужно прибавить ко всем элементам массива A на полуинтервале $[ql, qr)$ число x.
3. Даны ql, qr. Необходимо вернуть наибольший общий делитель (gcd, НОД) элементов массива A на полуинтервале $[ql, qr)$.

6.5.2 Решение + доказательство упрощенной версии задачи

Давайте сначала поймем, как решать эту задачу, если запросов первого типа нет. В отличие от других задач, это уже не так очевидно. Если ко всем числам на отрезке прибавили x, то не совсем понятно, как изменился НОД чисел на этом отрезке.

Как известно, $gcd(a,b) = gcd(a-b,b)$ (это факт, на котором основан алгоритм Евклида). То есть мы можем заменить одно из чисел на их разность. Но давайте вместо того, чтобы заменять, просто его добавим. Хуже не будет:

$$gcd(a,b) = gcd(a,b,a-b)$$

Этот факт можно обобщить для большего количества членов:

$$gcd(a,b,c) = gcd(a,b,c,a-b,b-c,c-a)$$

И так далее. То есть НОД чисел $a_1,a_2,\ldots,a_k$ равен НОДу этих чисел и всех их возможных $\frac{k\cdot(k-1)}{2}$ попарных разностей.

С другой стороны, давайте заметим, что из этих членов можно оставить только несколько. Нужно оставить всего k из них так, чтобы их линейными комбинациями можно было получить $a_1,a_2,\ldots,a_k$. И если эти числа можно получить линейными комбинациями, то НОД такого подмножества будет равен $gcd(a_1,a_2,\ldots,a_k)$.

Давайте представим все числа в виде графа. Ребро между двумя вершинами будет соответствовать их разности. Тогда на самом деле нам достаточно взять какое-то остовное дерево этого графа, а также одно любое из чисел $a_1,a_2,\ldots,a_k$. Почему? Пускай мы взяли число a_i, а также какое-то остовное дерево. Как получить линейную комбинацию, равную a_j? Так как мы взяли остовное дерево, то между a_i и a_j в этом дереве есть путь (все, что нам надо от дерева — это связность) $b_1,b_2,\ldots,b_l$, где $b_1 = a_i$ и $b_l = a_j$. Тогда в нашем множестве есть числа $b_1 = a_i$, $b_2 - b_1$, $b_3 - b_2$, $\ldots$, $b_l - b_{l-1}$. Их сумма как раз равна $b_l = a_j$. Что и требовалось доказать. Кроме того, если бы в нашем множестве не было числа a_i, а были бы только разности из какого-то остовного дерева, то исходные числа мы бы получить не смогли таким образом, но разность любых двух — без проблем. Нужно опять же просуммировать все разности на каком-то пути.

Для чего мы это доказывали? Давайте хранить на отрезке отдельно какое-то любое число с этого отрезка (`any_value`), а также НОД всех попарных разностей (`diff_gcd`), который, как мы уже выяснили, равен НОДу какого-то остовного дерева. И тогда НОД на отрезке будет вычисляться просто как gcd(`any_value`,`diff_gcd`). Однако заметим, что если мы поддерживаем такие значения в вершине, то прибавление числа — это уже не проблема. Если ко всем числам на отрезке прибавили константу y, то их попарные разности не поменялись, а значит, не поменялось и `diff_gcd`. А к `any_value`, так же, как и ко всем остальным значениям на отрезке, просто прибавится эта самая константа y.

Осталось только понять, как пересчитывать значения `any_value` и `diff_gcd` в вершине через значения в детях. `any_value` пересчитать очень легко. Можно просто взять любое из `any_value` для левого и правого поддеревьев. А как пересчитать `diff_gcd`? В `diff_gcd` левого поддерева хранится НОД какого-то остовного дерева левого подотрезка, а в `diff_gcd` правого поддерева хранится НОД какого-то остовного дерева правого подотрезка. Поэтому все, что нам нужно, — это соединить эти два остовных дерева, то есть добавить какое-то ребро. Но у нас как раз хранятся `any_value` для обоих поддеревьев. Возьмем их разность. То есть

```
diff_gcd[v] = gcd(diff_gcd[left_child], diff_gcd[right_child],
any_value[left_child] - any_value[right_child])
```

И тогда такое решение работает за $O(n\log C + q(\log n + \log C))$. Во втором слагаемом логарифмы складываются, а не перемножаются по той же причине, по которой так происходит в дереве отрезков с поиском *gcd* на отрезке и присваиванием на отрезке. Потому что последовательное вычисление НОД у k чисел, не больших C по модулю, работает за $O(k + \log C)$.

> **Упражнение 6.1** Докажите, что последовательное вычисление НОД k чисел, не больших C по модулю, работает за $O(k + \log C)$. ■

6.5.3 Решение + доказательство полной версии задачи

Итак, мы научились поддерживать операции `+=` и *gcd* на отрезке. Давайте добавим к этому еще `min=`.

Давайте немного изменим концепцию. Будем поддерживать на отрезке не НОД всех попарных разностей, а НОД всех попарных разностей чисел, не равных максимуму на отрезке. А максимумы (как и в Ji Driver Segment Tree) мы будем обрабатывать отдельно. Тогда при условии `tag_condition` (которое, как и `break_condition`, абсолютно такое же, как и в Ji Driver Segment Tree) мы сможем легко пересчитать значения. Нам нужно будет изменить только максимум, а `diff_gcd` никак не поменяется, потому что мы изменяем значения только максимумов, а они не входят в `diff_gcd`.

Как же нам тогда находить НОД на отрезке, зная эти значения? Все числа, не равные максимуму, уже объединены в остовное дерево внутри `diff_gcd`. Остается только присоединить максимумы (или один из них, потому что они все равно равны друг другу). В этом нам как раз может помочь `second_max`. Поэтому НОД на отрезке можно вычислить по такой формуле:

```
gcd(diff_gcd, max - second_max, max)
```

В оригинальной статье асимптотика этого алгоритма указана как $O(q\log^3 n)$. Там подразумевается, что $C = q = n$, так что в реальности такая оценка будет выглядеть как $O(n\log n\log C + q\log^2 n\log C)$.

Однако давайте улучшим эту оценку. Давайте докажем, что асимптотика на самом деле $O(n(\log n + \log C) + q\log n(\log n + \log C))$.

Почему мы не можем сказать так же, как и в облегченной версии задачи, что логарифмы складываются, а не перемножаются? Дело в том, что НОД k чисел, не больших C по модулю, вычисляется за $O(k + \log C)$ только в том случае, если мы вычисляем НОД последовательно. То есть сначала берем НОД двух чисел, потом берем НОД этого НОДа и следующего числа, и так далее. Именно так работает обычное дерево отрезков: в нем мы спускаемся вниз по левой и правой границам отрезка, то есть по бамбукам, в которых будет последовательно вычисляться НОД. Однако в Segment Tree Beats мы посещаем большое количество дополнительных вершин, которые вовсе не образуют бамбуки, так что логарифмы будут перемножаться. Но давайте копнем глубже.

Давайте вспомним, как мы оценивали асимптотику в задаче `min=`, `+=`. Мы ввели потенциал, равный количеству помеченных вершин в дереве, то есть тех вершин, у которых максимум в левом поддереве не совпадает с максимумом в правом поддереве. Мы доказали, что суммарно за все время он может увеличиться максимум на $O(n + q\log n)$, а также на каждые $O(\log n)$ посещенных дополнительных вершин этот потенциал уменьшается на 1, поэтому асимптотика будет $O(n\log n + q\log^2 n)$. Там мы показали, что в поддереве любой дополнительной вершины есть метка, которая удалится, поэтому если удалилось k меток, то было посещено не более $k\log n$ вершин. Однако нас интересуют не все вершины, а те, в которых происходит разветвление. Ведь в

бамбуке, как мы уже поняли, НОД вычисляется быстро. Но ведь все дополнительные вершины как раз таки лежат на k путях до корня от удаленных меток. И на каждом таком пути НОД будет вычисляться за $O(\log n + \log C)$, а так как всего за все время было удалено максимум $O(n + q \log n)$ меток, то асимптотика алгоритма будет равна $O(n(\log n + \log C) + q \log n(\log n + \log C))$. Что и требовалось доказать.

6.6 %=, = на отрезке, $\sum$

6.6.1 Формулировка

Простую версию этой задачи мы уже рассматривали ранее. В этой задаче у нас есть массив неотрицательных целых чисел A $(0 \leqslant A_i < C)$, а также имеются запросы трех типов:

1. Даны ql, qr, x $(1 \leqslant x < C)$. Нужно заменить все элементы массива A на полуинтервале $[ql, qr)$ на $A_i \bmod x$.
2. Даны ql, qr, y $(0 \leqslant y < C)$. Нужно заменить все элементы массива A на **полуинтервале** от ql до qr на y.
3. Даны ql, qr. Необходимо вернуть сумму элементов массива A на полуинтервале $[ql, qr)$.

6.6.2 Решение

Решение ничем не отличается от решения простой версии задачи, описанного ранее [3].

6.6.3 Доказательство

Спасибо Антону Степанову за доказательство

Однако теперь старый потенциал, равный сумме логарифмов элементов на отрезке, больше не работает, потому что при присвоении на отрезке он может очень сильно увеличиваться. Придумаем новый потенциал и докажем, что время работы останется все тем же самым: $O((n+q) \log n \log C)$.

Для начала скажем, что вершины, на отрезке которых все числа равны, являются помеченными, а потенциал этих вершин равен нулю. В любом случае, такая вершина не может быть дополнительной, так как в ней выполнится `tag_condition`, потому что $max = min$.

Отрезок, за который отвечает любая вершина разбивается на подотрезки с одинаковыми значениями. У любой непомеченной вершины таких отрезков не меньше двух. Если раньше мы считали в потенциале сумму логарифмов всех элементов на отрезке, то теперь подотрезок одинаковых значений в потенциале будет считаться за одно значение, то есть потенциал вершины — это сумма логарифмов элементов из подотрезков равных значений, на которые разбивается отрезок текущей вершины. Разумеется, чтобы у нас не было логарифма нуля, мы будем брать логарифмы элементов, увеличенных на 1.

Как уже было сказано ранее, потенциалы помеченных вершин при этом считаются равными нулю. Потенциал всего дерева как всегда равен сумме потенциалов всех вершин.

Наш новый потенциал не больше старого, так что очевидно, что изначально $\Phi \leqslant n \log n \log C$.

При посещении тупиковых вершин либо потенциал не меняется, если выполнилось `break_condition`, либо потенциал как был нулем, так и останется, если выполнилось `tag_condition`, потому что такие вершины помечены.

[3] 6.2

При посещении дополнительных вершин в запросе первого типа не появляется никаких новых отрезков значений. Только старые значения уменьшаются и, возможно, некоторые отрезки склеиваются, так что потенциал может только уменьшаться.

Вершины, которые отвечают за отрезки, лежащие полностью внутри отрезка изменения запроса второго типа, становятся помеченными, поэтому потенциал в них становится равен нулю.

При посещении же обычных вершин потенциалы могут увеличиваться, но не очень сильно, при запросах обоих типов. Могло появиться не более трех новых отрезков элементов, каждый из которых даст вклад $O(\log C)$ в потенциал. А если учесть, что обычных вершин в каждом запросе $O(\log n)$, можно понять, что потенциал за все время увеличится максимум на $O(q \log n \log C)$.

Остается лишь показать, что при посещении дополнительной вершины потенциал уменьшается хотя бы на 1. Действительно, не выполнилось ни `break_condition`, ни `tag_condition`, так что вершина не помечена и $max \geqslant x$, поэтому на отрезке max значение уменьшится хотя бы в два раза после взятия по модулю, поэтому потенциал текущей вершины уменьшится хотя бы на 1. Что и требовалось доказать.

Таким образом, асимптотика получившегося алгоритма — $O((n+q)\log n \log C)$.

6.7 $\sqrt{}=$, +=, $\sum$, max, min

6.7.1 Формулировка

В этой задаче у нас есть массив неотрицательных целых чисел A $(0 \leqslant A_i < C)$, а также имеются запросы пяти типов:

1. Даны ql, qr. Нужно заменить все элементы массива A на полуинтервале $[ql, qr)$ на $\lfloor \sqrt{A_i} \rfloor$.
2. Даны ql, qr, x $(0 \leqslant x < C)$. Нужно прибавить ко всем элементам массива A на полуинтервале $[ql, qr)$ число x.
3. Даны ql, qr. Необходимо вернуть сумму элементов массива A на полуинтервале $[ql, qr)$.
4. Даны ql, qr. Необходимо вернуть максимум элементов массива A на полуинтервале $[ql, qr)$.
5. Даны ql, qr. Необходимо вернуть минимум элементов массива A на полуинтервале $[ql, qr)$.

Эта задача доступна здесь или здесь. Это китайские сайты, поэтому там не так просто зарегистрироваться. На этих сайтах нет двух последних операций, но они поддерживаются абсолютно очевидно и все равно необходимы во время решения, так что это ничего не меняет.

6.7.2 Решение

Мы будем поддерживать в каждой вершине сумму, максимум и минимум. Все запросы кроме первого типа выполняются так же, как и в обычном дереве отрезков.

Кроме того, для первой операции нам понадобится еще уметь присваивать на отрезке, поэтому в каждой вершине хранится 2 `push`-а: что надо присвоить на отрезке и что надо прибавить на отрезке. При этом только один из них может быть в вершине, потому что комбинация присвоения и прибавления — это то же самое, что просто присвоение.

`break_condition` в этом случае является стандартным:

`break_condition = qr <= l || r <= ql`.

А каким же должно быть `tag_condition`? Первая идея, которая приходит в голову — это

`tag_condition = ql <= l && r <= qr &&` $\lfloor\sqrt{\max}\rfloor = \lfloor\sqrt{\min}\rfloor$,

потому что в этом случае корни из всех чисел на отрезке равны, и нужно просто произвести присвоение этому корню на этом отрезке. **Однако это тот случай, когда очевидный вариант не работает!** Представим себе ситуацию, в которой изначально массив имеет вид $1,2,1,2,1,2,\dots,1,2$. После чего к нему $\frac{q}{2}$ раз применяют две операции: $+=2$ на всем массиве и $\sqrt{}=$ на всем массиве. После первой операции массив превращается в $3,4,3,4,3,4,\dots,3,4$. А после второй он возвращается в исходное состояние. Однако заметим, что целая часть от корня из 3 — это 1, а целая часть от корня из 4 — это 2, поэтому ни в одной вершине кроме листьев `tag_condition` не выполнится, так что каждая операция $\sqrt{}=$ будет выполняться за $O(n)$, и итоговая асимптотика будет $O(q\cdot n)$. Поэтому это `tag_condition` нам не подходит.

На самом деле, правильный `tag_condition` выглядит следующим образом:

`tag_condition = ql <= l && r <= qr && max[v] - min[v] <= 1`.

Казалось бы, условие стало только слабее. Раньше мы проверяли, что корни совпадают, а теперь мы проверяем, что либо максимум равен минимуму, либо они отличаются на 1. Но здесь кроется принципиальная разница. Единственный плохой случай — это когда `max` — это квадрат, а минимум на 1 меньше. Тогда при взятии корня разница между ними останется равной 1, и операцией `+=` можно будет вернуть массив в исходное положение.

Как же нам обновить значение в вершине, когда выполнилось `tag_condition`? Максимум и минимум, очевидно, заменяются на свои корни. А как поменяется сумма? Если целые части корней из минимума и максимума равны, то после применения операции на отрезке все числа будут равны, поэтому нужно просто присвоить $\lfloor\sqrt{\max}\rfloor$ на отрезке. Это частный случай неправильного `tag_condition`, который мы обсуждали ранее.

Остается один случай: если целые части корней из минимума и максимума не совпадают. При этом максимум и минимум отличаются в точности на 1. Тогда и корни тоже будут отличаться в точности на 1. Максимум был равен k^2, а минимум k^2-1. При этом k^2 заменился на k, а k^2-1 заменился на $k-1$. Поэтому нужно просто ко всем числам на отрезке прибавить $k-k^2$.

6.7.3 Доказательство

Давайте докажем, что с таким `tag_condition` асимптотика будет $O(n\log C + q\log n\log C)$.

Давайте введем потенциал вершины $\varphi(v) = \log($`max[v] - min[v]` $+1)$. Прибавление единицы опять же нужно только для того, чтобы не брать логарифм нуля в случае, когда максимум равен минимуму. Общим потенциалом Φ будет сумма потенциалов всех вершин дерева.

Заметим, что потенциал любой вершины неотрицателен и не превышает $\log C$, так что в любой момент времени верно $0 \leqslant \Phi \leqslant O(n\log C)$.

При этом как может увеличиваться этот потенциал? В запросах типа `get` элементы массива не меняются, так что потенциал тоже не меняется. Для запросов второго типа потенциал вершины мог поменяться только в том случае, если данная вершина пересекается с отрезком запроса, но не лежит в нем полностью, потому что если она лежит в нем полностью, то и к минимуму, и к максимуму прибавится x, так что разность не поменяется. А вершин, которые пересекаются с отрезком запроса, но не лежат в нем полностью, $O(\log n)$ штук, так что потенциал может увеличиться максимум на $O(q\log n\log C)$ за все время.

Аналогично, операция $\sqrt{}=$ не может увеличить потенциал, если вершина полностью лежит в отрезке запроса, потому что разность корней не больше разности исходных чисел (это легко проверить, но дальше мы докажем даже более сильное условие). А вершин, которые пересекаются с запросом, но не лежат в нем полностью, опять же $O(\log n)$ штук, так что опять же увеличение за все время — это $O(q \log n \log C)$.

Осталось понять, что при посещении дополнительной вершины потенциал уменьшается хотя бы на $\log(1.5)$ (не пугайтесь этого числа, на самом деле неважно, чему оно равно, главное, что это положительная константа), тогда итоговая асимптотика будет равна $O(n \log C + q \log n \log C)$. В этом нам поможет следующий факт:

Теорема 6.7.1 Если a и b — неотрицательные целые числа, и $a \geqslant b+2$, то $a-b \geqslant 1.5 \cdot \left(\lfloor\sqrt{a}\rfloor - \left\lfloor\sqrt{b}\right\rfloor\right) + 0.5$.

Доказательство. Обозначим $\left\lfloor\sqrt{b}\right\rfloor = m$ и $\lfloor\sqrt{a}\rfloor = n+m$. При этом $n \geqslant 0$, потому что $a > b$. Тогда $b = m^2 + l$, где $0 \leqslant l \leqslant 2 \cdot m$ и $a = (n+m)^2 + k$, где $0 \leqslant k \leqslant 2 \cdot (n+m)$. Разберем два случая:

1. $n \leqslant 1$. То есть $\lfloor\sqrt{a}\rfloor - \left\lfloor\sqrt{b}\right\rfloor \leqslant 1$, так что неравенство, которое нам надо доказать, превращается в $a - b \geqslant 1.5 \cdot 1 + 0.5 = 2$. Это верно из-за условия на то, что $a \geqslant b+2$.
2. $n \geqslant 2$. В этом случае мы знаем, что $a \geqslant (n+m)^2$ и $b \leqslant m^2 + 2m$, поэтому $a - b \geqslant (n+m)^2 - (m^2+2m) = n^2 + 2nm - 2m = n^2 + 2m \cdot (n-1) \geqslant n^2$. Последнее неравенство верно, потому что все множители во втором слагаемом неотрицательны. Нам нужно доказать, что это не меньше, чем $1.5 \cdot n + 0.5$. Это легко проверить, потому что $n^2 \geqslant 2n = 1.5n + 0.5n \geqslant 1.5n + 0.5$. Первое неравенство верно из-за того, что $n \geqslant 2$, а второе из-за того, что $n \geqslant 1$. Что и требовалось доказать.

■

Давайте немного преобразуем получившееся неравенство. Прибавим к обеим частям 1:

$$a - b + 1 \geqslant 1.5\left(\lfloor\sqrt{a}\rfloor - \left\lfloor\sqrt{b}\right\rfloor + 1\right)$$

И возьмем логарифмы от обеих частей:
$\log(a-b+1) \geqslant \log\left(1.5\left(\lfloor\sqrt{a}\rfloor - \left\lfloor\sqrt{b}\right\rfloor + 1\right)\right) = \log(1.5) + \log\left(\lfloor\sqrt{a}\rfloor - \left\lfloor\sqrt{b}\right\rfloor + 1\right)$.

То есть мы доказали, что потенциал уменьшился хотя бы на $\log(1.5)$, и асимптотика решения доказана.

Замечание 6.7.2 Обратите внимание, что операция присвоения на отрезке будет менять потенциалы так же несильно, как и операция прибавления на отрезке, так что ее мы тоже можем поддерживать. Кроме того, мы и так присваиваем на отрезке в одном из случаев, когда берем корень, поэтому особо ничего нового к решению добавлять не придется.

6.8 /=, +=, ∑, max, min

6.8.1 Формулировка

В этой задаче у нас есть массив неотрицательных целых чисел A $(0 \leqslant A_i < C)$, а также имеются запросы пяти типов:

1. Даны ql, qr, x $(x \geqslant 1)$. Нужно поделить все элементы массива A на полуинтервале $[ql, qr)$ на x нацело.

2. Даны ql, qr, y $(0 \leqslant y < C)$. Нужно прибавить ко всем элементам массива A на полуинтервале $[ql, qr)$ число y.
3. Даны ql, qr. Необходимо вернуть сумму элементов массива A на полуинтервале $[ql, qr)$.
4. Даны ql, qr. Необходимо вернуть максимум элементов массива A на полуинтервале $[ql, qr)$.
5. Даны ql, qr. Необходимо вернуть минимум элементов массива A на полуинтервале $[ql, qr)$.

6.8.2 Решение

Как бы это ни было удивительно, но решение абсолютно идентично решению предыдущей задачи.

```
break_condition = qr <= l || r <= ql
tag_condition = ql <= l && r <= qr && max[v] - min[v] <= 1
```

Есть только одно новое условие: если мы пытаемся поделить на отрезке на единицу, то мы просто проигнорируем этот запрос, потому что деление на 1 не меняет элементов.

И в случае, если `tag_condition` выполнилось, мы делаем точно такие же изменения, как и для корня. Если $\left\lfloor \frac{\max}{x} \right\rfloor = \left\lfloor \frac{\min}{x} \right\rfloor$, то нужно всем числам на отрезке присвоить $\left\lfloor \frac{\max}{x} \right\rfloor$, а если $\left\lfloor \frac{\max}{x} \right\rfloor \neq \left\lfloor \frac{\min}{x} \right\rfloor$, то тогда $\max = kx$ для некоторого k и $\min = kx - 1$. При этом $\left\lfloor \frac{\max}{x} \right\rfloor = k$ и $\left\lfloor \frac{\min}{x} \right\rfloor = k - 1$, так что надо просто ко всем числам на отрезке прибавить $k - kx$.

6.8.3 Доказательство

Почему же такое решение будет работать?

Проведем абсолютно такое же доказательство с таким же потенциалом. Для всех операций кроме первой ничего не поменялось. Для первой операции опять же потенциал вершины не мог увеличиться, если вершина полностью лежит в отрезке запроса, потому что после деления разность максимума и минимума не могла увеличиться (это несложно проверить, но далее мы докажем более сильное условие).

Так что все, что нам надо доказать, — это то, что при посещении дополнительной вершины потенциал уменьшится хотя бы на $\log(\frac{4}{3})$ (как и раньше, просто положительная константа), и тогда асимптотика будет равна $O(n \log C + q \log n \log C)$. В этот момент нам как раз пригодится то, что мы игнорируем запросы, в которых $x = 1$, потому что в этом случае никакие потенциалы не меняются, и мы бы просто делали лишние действия.

Теорема 6.8.1 Если a, b и x — неотрицательные целые числа, при этом $a \geqslant b + 2$ и $x \geqslant 2$, то $a - b \geqslant \frac{4}{3} \cdot \left(\left\lfloor \frac{a}{x} \right\rfloor - \left\lfloor \frac{b}{x} \right\rfloor \right) + \frac{1}{3}$.

Доказательство. Обозначим $\left\lfloor \frac{b}{x} \right\rfloor = m$ и $\left\lfloor \frac{a}{x} \right\rfloor = n + m$. При этом $n \geqslant 0$, потому что $a > b$. Тогда $b = mx + l$, где $0 \leqslant l < x$ и $a = (n+m) \cdot x + k$, где $0 \leqslant k < x$. Разберем два случая:

1. $n \leqslant 1$. То есть $\left\lfloor \frac{a}{x} \right\rfloor - \left\lfloor \frac{b}{x} \right\rfloor \leqslant 1$, так что неравенство, которое нам надо доказать, превращается в $a - b \geqslant \frac{4}{3} \cdot 1 + \frac{1}{3} = \frac{5}{3}$. Это верно из-за условия на то, что $a \geqslant b + 2$.
2. $n \geqslant 2$. В этом случае мы знаем, что $a \geqslant (n+m) \cdot x$ и $b \leqslant mx + x - 1$, поэтому $a - b \geqslant (n+m) \cdot x - (mx + x - 1) = nx - x + 1$. И мы хотим доказать, что это не меньше, чем $\frac{4}{3}n + \frac{1}{3}$. Перенесем все из правой части в левую, а x перенесем в правую:

$$n \cdot \left(x - \frac{4}{3} \right) + \frac{2}{3} \geqslant x$$

$n \geqslant 2$, поэтому заменим в левой части:

$$2 \cdot \left(x - \frac{4}{3}\right) + \frac{2}{3} \geqslant x$$

Если раскрыть левую часть, получится:

$$2x - 2 \geqslant x$$

Это верно в силу того, что $x \geqslant 2$. Что и требовалось доказать.

Давайте немного преобразуем получившееся неравенство. Прибавим к обеим частям 1:

$$a - b + 1 \geqslant \frac{4}{3}\left(\left\lfloor \frac{a}{x} \right\rfloor - \left\lfloor \frac{b}{x} \right\rfloor + 1\right)$$

И возьмем логарифмы от обеих частей:

$\log(a-b+1) \geqslant \log\left(\frac{4}{3}\left(\left\lfloor \frac{a}{x} \right\rfloor - \left\lfloor \frac{b}{x} \right\rfloor + 1\right)\right) = \log\left(\frac{4}{3}\right) + \log\left(\left\lfloor \frac{a}{x} \right\rfloor - \left\lfloor \frac{b}{x} \right\rfloor + 1\right)$.

То есть мы доказали, что потенциал уменьшился хотя бы на $\log(\frac{4}{3})$, и асимптотика решения доказана.

■

6.9 &=, |=, max

6.9.1 Формулировка

В этой задаче у нас есть массив целых чисел A $(0 \leqslant A_i < C)$, а также имеются запросы трех типов:

1. Даны ql, qr, x $(0 \leqslant x < C)$. Нужно заменить все элементы массива A на полуинтервале $[ql, qr)$ на $A_i \& x$ (побитовое «и»).
2. Даны ql, qr, x $(0 \leqslant y < C)$. Нужно заменить все элементы массива A на полуинтервале $[ql, qr)$ на $A_i | x$ (побитовое «или»).
3. Даны ql, qr. Необходимо вернуть максимум элементов массива A на полуинтервале $[ql, qr)$.

В данном случае стоит считать, что $C = 2^k$ для некоторого натурального k, то есть мы работаем с k-битными числами.

Эта задача доступна здесь.

6.9.2 Решение

Давайте заметим, что & — это то же самое, что побитовый минимум, а | — побитовый максимум, так что эта задача — это в каком-то смысле вариация Ji Driver Segment Tree для битовых операций.

Давайте хранить в каждой вершине дерева максимум на соответствующем подотрезке, побитовое «или», побитовое «и», а также ленивые обновления типа `pushAnd` и `pushOr`, которые означают, что ко всем числам на отрезке нужно применить операцию «и», а также «или». Можно заметить, что даже если к нам приходят много операций первых двух типов, мы все равно можем скомбинировать их в два таких значения, которые нужно проталкивать.

Теперь осталось понять, как это все считать, и какие будут `break_condition` и `tag_condition`. В данном случае нестандартным будет только `tag_condition`. Почему нам вообще не подходит обычное дерево отрезков? Потому что если ко всем числам на отрезке применить запрос одного из первых двух типов, то их относительных порядок может сильно поменяться, а значит, поменяется и максимум. Но в каком

случае относительный порядок не поменяется? Посмотрим для примера на первую операцию. Какими свойствами должны обладать числа $a_1 \leqslant a_2 \leqslant \ldots \leqslant a_k$, чтобы при этом выполнялось $a_1 \& x \leqslant a_2 \& x \leqslant \ldots \leqslant a_k \& x$? Те биты, которые в x установлены в единицу, не поменяются в числах. Поменяются только те биты, которые у a_i установлены в 1, а у x — в 0. И если у какого-то a_i была установлена 1, а у другого 0, то их относительный порядок мог поменяться. Но если на всех позициях, в которых у числа x стоит ноль, у всех a_i стоят одинаковые биты, то относительный порядок никак не изменится. Аналогичное утверждение можно сформулировать для операции «или»: если на всех позициях, в которых у числа y стоит единица, у всех a_i выставлены одинаковые биты, то их относительный порядок не изменится.

А проверить, что на этих позициях у всех чисел стоят одинаковые биты, очень легко. Достаточно посмотреть на побитовое «и» и побитовое «или» всех чисел на отрезке. Тогда если какой-то бит у них совпадает, это как раз равносильно тому, что у всех чисел на отрезке этот бит совпадает. Так что `tag_condition` для первой и второй операции соответственно будут выглядеть следующим образом:

```
tag_condition_and = ql <= l && r <= qr && ((and[v] ^ or[v]) & ~x) == 0
tag_condition_or = ql <= l && r <= qr && ((and[v] ^ or[v]) & y) == 0
```

Упражнение 6.2 Хорошим упражнением будет понять, почему эта битовая магия соответствует именно тем условиям, которые мы описали ранее. ■

6.9.3 Доказательство

Почему же такое решение будет работать быстро?

Давайте введем потенциал $\varphi(v)$ вершины v дерева отрезков, который будет равен количеству битов, в которых не все числа на соответствующем отрезке совпадают, то есть, иными словами, `__builtin_popcount(and[v] ^ or[v])`. Потенциал всего дерева Φ, как обычно, будет равен сумме потенциалов всех вершин.

Заметим, что $0 \leqslant \varphi(v) \leqslant \log C$, потому что у нас есть всего $\log C$ битов, так что отличающихся не больше. Поэтому в любой момент времени $0 \leqslant \Phi \leqslant n \log C$.

В каких случаях потенциал может увеличиваться? Операция третьего типа не меняет элементов массива, так что потенциал тоже не меняется. Если у всех чисел на каком-то отрезке какой-то бит был равен, и мы применили ко всем этим числам операцию одного из первых двух типов, то этот бит не мог перестать быть равен. Так что увеличения потенциала могли происходить только в вершинах, подотрезок которых пересекается, но не лежит полностью в отрезке запроса, то есть в обычных вершинах. При этом для каждой такой вершины потенциал мог увеличиться максимум на $\log C$, а для каждого запроса таких вершин $O(\log n)$, так что можно сказать, что $\Phi_+ \leqslant O(q \log n \log C)$.

Теперь поймем, почему при посещении дополнительных вершин потенциал уменьшается. Если мы посетили какую-то дополнительную вершину, то существует такой бит, что на данный момент не у всех чисел он одинаковый, и при этом у x он выставлен в 0 (либо у y в 1 для запросов второго типа), то есть после применения текущего запроса у всех чисел этот бит будет равен нулю (или единице для запросов второго типа). Так что потенциал текущей вершины уменьшится как минимум на 1. Что и требовалось доказать.

Поэтому асимптотика получившегося алгоритма — $O(n \log C + q \log n \log C)$.

6.10 `min=`, `+=`, `max` over $\sum$ для нескольких массивов параллельно

6.10.1 Формулировка

В этой задаче у нас есть сразу **два массива** целых чисел A и B, а также имеются запросы пяти типов:

1. Даны ql, qr, x. Нужно заменить все элементы массива a на полуинтервале $[ql, qr)$ на $\min(A_i, x)$.
2. Аналогичный запрос для массива B.
3. Даны ql, qr, y. Нужно прибавить ко всем элементам массива A на полуинтервале $[ql, qr)$ число y.
4. Аналогичный запрос для массива B.
5. Даны ql, qr. Необходимо вернуть $\max_{ql \leqslant i < qr} A_i + B_i$.

Последний запрос как раз называется max over $\sum$, то есть максимум от поэлементной суммы двух массивов.

Также в конце решения мы покажем, как это обобщается на случай бо́льшего количества массивов.

6.10.2 Решение + доказательство

Если бы у нас не было операции `min=`, то эта задача была бы весьма простой. Мы бы пользовались обычным деревом отрезков, и если мы на отрезке прибавляем число в одном из массивов, то к максимуму поэлементной суммы массивов тоже прибавится это самое число. Иными словами, можно считать, что у нас есть всего один массив C, определенный по правилу $C_i = A_i + B_i$, и операции производятся с ним.

Однако, если у нас есть операция `min=`, то нам важны значения каждого из массивов по отдельности.

Мы воспользуемся стандартной техникой из Ji Driver Segment Tree, то есть будем хранить максимум, их количество и второй максимум. Тогда при условии `tag_condition` мы меняем в массиве какой-то разрозненный набор позиций, на которых стоят максимумы. Как в таком случае пересчитать максимум поэлементной суммы массивов? Давайте разделим позиции на подотрезке, которому соответсвует текущая вершина, на 4 типа:

1. В обоих массивах на этой позиции стоят максимумы на отрезке
2. В первом массиве на этой позиции стоит максимум на отрезке, а во втором — нет
3. Во втором массиве на этой позиции стоит максимум на отрезке, а в первом — нет
4. В обоих массивах на этой позиции стоят не максимумы на отрезке

И после такого разделения выясняется, что хранить максимум сумм для каждого из четырех типов по отдельности оказывается очень просто. Если выполнилось `tag_condition`, то нужно поменять значения, соответствующие тем типам, в которых в этом массиве стоит максимум. При этом значение максимума изменилось с `max` на x, поэтому из соответствующих значений как раз надо отнять `max`$-x$. Ответом на запрос тогда будет просто максимум из четырех отдельных максимумов для каждого из типов.

Осталось научиться пересчитывать значения в вершине через значения в детях. Но это тоже делается очень просто. Мы сначала пересчитываем максимумы в обоих массивах в текущей вершине, после чего для каждого типа каждого из детей понимаем, стоят ли там максимумы или нет. Если там был не максимум для сына, то это будет не максимум и для отца; а если это был максимум для сына, то он мог как остаться максимумом, так и перестать им быть.

Решение ничем не отличается от задачи `max=`, `+=`, мы просто поддерживаем в вершине четыре дополнительных значения. Поэтому асимптотика получается $O(n \log n + q \log^2 n)$.

> **Замечание 6.10.1** Если подумать, можно заметить, что эта идея легко обобщается на большее количество массивов. Если у нас k массивов, то мы будем поддерживать 2^k дополнительных величин в каждой вершине, потому что в каждом массиве на каждой позиции может стоять либо максимум, либо не максимум (2 варианта), и можно брать всевозможные комбинации максимумов и не максимумов для разных массивов. Получается всего 2^k различных комбинаций. И асимптотика тогда возрастет до $O((n\log n + q\log^2 n)\cdot 2^k)$.

6.11 Историческая информация: количество изменений элемента

6.11.1 Формулировка

В этой задаче у нас есть массив целых чисел A, а также имеются запросы четырех типов:

1. Даны ql, qr, x. Нужно заменить все элементы массива A на полуинтервале $[ql, qr)$ на $\min(A_i, x)$.
2. Даны ql, qr, y. Нужно заменить все элементы массива A на полуинтервале $[ql, qr)$ на $\max(A_i, y)$.
3. Даны ql, qr, z. Нужно прибавить ко всем элементам массива A на полуинтервале $[ql, qr)$ число z.
4. Даны ql, qr. Необходимо вернуть сумму элементов массива S на полуинтервале $[ql, qr)$, где S_i — это количество раз, когда i-й элемент массива A менялся.

6.11.2 Решение + доказательство

Эта задача — первый базовый пример того, что такое «историческая информация». Обычно мы делаем запросы к элементам массива на данный момент, и мы не заботимся о том, как текущий массив был получен. Историческая информация же хранит в себе знания о том, как наш массив менялся с течением времени, а не только его финальный вид.

Однако нет ничего сложного в том, чтобы поддерживать эту информацию. Если бы у нас была только операция `+=`, то каждый раз, когда массив A меняют на полуинтервале $[ql, qr)$, к массиву S надо было бы на этом полуинтервале просто прибавить 1, потому что при каждой операции меняются абсолютно все элементы на подотрезке. Однако когда появляется операция `min=`, это уже неверно. Но все еще мы можем понять, сколько элементов поменялось. Мы останавливаем рекурсию в тот момент, когда выполнилось `tag_condition`. Это значит, что меняются только максимумы. Поэтому изменится ровно `cnt_max` элементов. На асимптотику алгоритма это никак влиять не будет, она все еще будет равна $O(n\log n + q\log^2 n)$. Стоит помнить о том, что если мы храним отложенную информацию о двух операциях `min=`, то элементы ниже поменялись дважды.

Далее мы рассмотрим более сложные версии исторической информации.

6.12 Историческая информация: исторический максимум

6.12.1 Формулировка

В этой задаче у нас есть массив целых чисел A, а также имеются запросы трех типов:

1. Даны ql, qr, x. Нужно заменить все элементы массива A на полуинтервале $[ql, qr)$ на $\min(A_i, x)$.
2. Даны ql, qr, y. Нужно прибавить ко всем элементам массива A на полуинтервале $[ql, qr)$ число y.

3. Даны ql, qr. Необходимо вернуть сумму элементов массива M на полуинтервале $[ql, qr)$, где M_i — это исторический максимум A_i, то есть самое большое значение, которое хранилось в A_i за все время.

Эта задача доступна здесь в немного ином виде. Там вместо максимумов минимумы, и третий запрос берет не сумму элементов массива M, а максимум.

Также похожая задача есть здесь.

6.12.2 Решение + доказательство

Первые две операции мы поддерживаем как обычно. И асимптотика получившегося алгоритма будет опять же $O(n \log n + q \log^2 n)$. Остается понять, как поддерживать сумму на отрезке массива M.

Давайте введем дополнительный массив D, элементы которого будут определяться по формуле $D_i = M_i - A_i$. А массив M мы на самом деле не будем нигде поддерживать. Действительно, заметим, что

$$\sum_{ql \leqslant i < qr} M_i = \sum_{ql \leqslant i < qr} ((M_i - A_i) + A_i) = \sum_{ql \leqslant i < qr} (D_i + A_i) = \sum_{ql \leqslant i < qr} D_i + \sum_{ql \leqslant i < qr} A_i$$

Поэтому нам нужно уметь отдельно поддерживать сумму на отрезках массивов A и D, и с их помощью мы сможем находить сумму на отрезке массива M. В этом и заключается основная идея исторического максимума. Но почему пересчитывать массив D проще, чем массив M?

Давайте сначала рассмотрим, что произойдет с массивом D после операции второго типа. К элементам массива A прибавится x, поэтому из D нужно вычесть x. Однако если A_i стало новым историческим максимумом, то M_i тоже изменится, и станет равно A_i. Это происходит ровно в тот момент, когда $M_i - A_i$ становится отрицательным, и в этот момент нужно заменить D_i на ноль. То есть на самом деле D_i заменяется на $\max(D_i - x, 0)$. А эту операцию можно разделить на две: сначала вычесть на отрезке, а потом применить `max=` на отрезке с нулем. Это все мы умеем делать.

Теперь разберемся, что происходит с массивом D при операции `min=`. В этой операции элементы могут только убывать, поэтому M_i точно не поменяются. А A_i поменяются весьма понятным образом. Когда мы пришли в вершину, в которой выполнилось `tag_condition`, D_i поменяется точно так же, как и A_i, а далее это изменение надо будет просто протолкнуть в детей.

6.13 Историческая информация: историческая сумма

6.13.1 Формулировка

В этой задаче у нас есть массив целых чисел A, а также имеются запросы двух типов:

1. Даны ql, qr, x. Нужно прибавить ко всем элементам массива A на полуинтервале $[ql, qr)$ число x.
2. Даны ql, qr. Необходимо вернуть сумму элементов массива S на полуинтервале $[ql, qr)$, где S_i — сумма элементов, стоящих на позиции i в массиве A за все время. То есть после каждой операции к S_i прибавляется A_i для всех позиций i.

6.13.2 Решение + доказательство

Эта задача похожа на предыдущую, но на этот раз вместо исторического максимума мы поддерживаем историческую сумму. В этой задаче нам хватит на самом деле даже обычного дерева отрезков.

Введем дополнительный массив D, элементы которого будут определяться по формуле $D_i = S_i - ind \cdot A_i$, где ind — это количество операций, которые уже были произведены с массивом на данный момент (после каждой операции ind увеличивается на 1).

Опять же, сам массив S мы нигде поддерживать не будем, потому что

$$\sum_{ql \leqslant i < r} S_i = \sum_{ql \leqslant i < r} (D_i + ind \cdot A_i) = \sum_{ql \leqslant i < r} D_i + ind \cdot \sum_{ql \leqslant i < r} A_i$$

Так что поддерживая сумму на отрезке в массивах D и A, мы сможем вычислять сумму на отрезке в массиве S. Посмотрим, как меняются элементы массива D при операции первого типа. Пускай до этой операции у нас были массивы S, A и D, а после применения операции они превратились в массивы S', A' и D'. Мы хотим научиться вычислять D'. Мы знаем, что $D_i = S_i - ind \cdot A_i$ и $D'_i = S'_i - (ind+1) \cdot A'_i$, потому что после применения операции ind увеличился на 1. Давайте раскроем последнюю формулу:

$$D'_i = S'_i - (ind+1) \cdot A'_i = (S_i + A'_i) - (ind+1) \cdot A'_i =$$

$$= S_i - ind \cdot A'_i = S_i - ind \cdot (A_i + x) = S_i - ind \cdot A_i - ind \cdot x = D_i - ind \cdot x$$

Так что у массиву D тоже просто прибавляется константа $(-ind \cdot x)$ на отрезке. Для элементов, которые не меняются, к S_i прибавится A_i и ind увеличится на 1, поэтому D_i никак не поменяются.

6.14 Задачи для практики

- Первая задача из этой главы. $\% =$ на отрезке, присвоение в точке и поиск суммы на отрезке.
- Ji Driver Segment Tree ($\min =$ на отрезке и поиск суммы на отрезке) можно решить здесь и здесь.
- Кроме того есть не совсем очевидное применение Ji Driver Segment Tree в задаче «Нагайна».
- Задачу с операцией $\sqrt{} =$ на отрезке можно решить здесь или здесь.
- Задачу с операцией $/ =$ на отрезке можно решить здесь.
- Задача с операциями $\& =$ и $| =$ на отрезке и поиском максимума на отрезке доступна здесь.
- Задача на сумму исторических максимумов доступна здесь. Также есть похожая задача здесь.
- Кроме того можете прорешать специально подготовленный контест на codeforces. Если у вас нет доступа к соревнованию, нужно сначала вступить в группу.

7. Алгоритм Фараха-Колтона и Бендера

Задача RMQ (Range Minimum Query) состоит в том, что на вход дан массив длины n и q запросов (отрезков этого массива), и необходимо для каждого запроса найти минимум на соответствующем отрезке массива. Есть много подходов к решению этой задачи. Есть тупое решение, которое работает за $O(n \cdot q)$, есть решение деревом отрезков, работающее за $O(n+q\log n)$, есть решение при помощи разреженных таблиц, работающее за $O(n\log n+q)$, и так далее. В этой статье мы рассмотрим подход, который нужен для этой конкретной задачи (дерево отрезков, к примеру, применимо для практически любых операций) и при этом работает за линейное время, а также альтернативный подход, который одновременно легче в реализации и быстрее на практике.

Стоит обратить внимание на то, что данная задача может ставиться в двух формулировках: offline и online. Offline означает, что все запросы даны заранее, и можно находить на них ответы в произвольном порядке. Однако в online версии задачи следующий запрос дается только после ответа на предыдущий. Мы рассмотрим обе версии задачи.

Кроме того, эту задачу можно делить на static и dynamic версии. В static версии массив фиксирован, а в dynamic версии могут приходить запросы изменения массива. Мы будем рассматривать именно static версию задачи.

7.1 RMQ offline. Вариация алгоритма Тарьяна за $O(\alpha(n))$ на запрос.

Ознакомиться с алгоритмом можно в предыдущей главе 3.

С первого взгляда это может показаться не очень ясным, но данный алгоритм имеет очень интересную связь с оригинальным алгоритмом Тарьяна для поиска LCA в дереве. Более глубокая связь между этими двумя алгоритмами станет понятна, когда мы разберем алгоритм Фараха-Колтона и Бендера (ФКБ), а пока можете воспринимать данный алгоритм как самостоятельный.

7.2 RMQ online. Улучшаем разреженные таблицы.

Разреженные таблицы хорошо подходят для решения задачи RMQ, потому что отвечают на запрос за константное время. Однако проблема в том, что на их построение уходит $O(n\log n)$ времени и памяти. В дальнейшем мы будем пытаться бороться с этим.

Давайте применим идею, схожую с идеей корневой декомпозиции. Разобьем массив на блоки. Однако длина блока будет не $\sqrt{n}$, а порядка $\log n$. Тогда количество блоков будет равно $\frac{n}{\log n}$. Давайте на каждом блоке за линейное время найдем минимум и на получившемся сжатом массиве построим разреженную таблицу. Ее построение займет $O\left(\frac{n}{\log n}\log\left(\frac{n}{\log n}\right)\right) = O\left(\frac{n}{\log n}\log n\right) = O(n)$ времени. Теперь мы умеем отвечать на запросы поиска минимума в том случае, если концы отрезка совпадают с концами блоков. Но что делать, если это не так?

Так же, как и в случае корневой декомпозиции, можно заметить, что любой отрезок запроса разбивается на несколько целых блоков, а также по краям слева и справа будут нецелые блоки. Для целых блоков мы можем найти минимум, используя нашу разреженную таблицу, остается найти минимум на маленьких кусочках слева и справа. Можно просто пройтись по ним и насчитать минимум. Так как длина блока — $O(\log n)$, то асимптотика ответа на запрос будет $O(\log n)$, но можно сделать и лучше.

Давайте заметим, что нам надо уметь искать минимумы на префиксах и суффиксах блоков. Все эти значения можно предпосчитать заранее за $O(n)$ (для каждого элемента массива сохранить минимум до конца и до начала его блока), тогда на запросы мы будем отвечать за $O(1)$, ведь минимум на отрезке — это минимум из трех величин: минимум на суффиксе левого неполного блока, минимум в разреженной таблице по целым блокам и минимум на префиксе правого неполного блока.

> **Упражнение 7.1** Почему же на этом статья не заканчивается, а только начинается? Стоит задуматься над этим вопросом, чтобы проверить свое понимание происходящего. ■

На самом деле, к сожалению, этого не достаточно. И проблема находится там, где ее не ждали: в маленьких отрезках. Действительно, если длина отрезка запроса не меньше логарифма, то он обязательно состоит из нескольких (возможно, нуля) целых блоков, одного префикса и одного суффикса. Но для маленьких отрезочков это может быть неверно в том случае, если отрезок целиком лежит внутри блока. Тогда он не разбивается на префикс и суффикс, да и внутренних целых блоков у него тоже нет.

Стоит помнить о том, что все последующие манипуляции, которые мы будем производить, нужны именно для того, чтобы побороть маленькие отрезки. Уже сейчас мы можем просто пройтись по этим отрезкам за линию и найти минимум, тогда для больших отрезков мы будем отвечать на запрос за $O(1)$, а для маленьких за $O(\log n)$, потому что их длина не больше логарифма.

Можно пойти с другой стороны и заметить, что маленьких отрезков не больше $n\log n$ штук (для каждой левой границы существует $\leqslant \log n$ маленьких отрезков), поэтому для них всех можно предпосчитать ответы заранее за $O(n\log n)$ и отвечать на запросы за $O(1)$, однако мы опять упираемся в проблему: либо в построении вылезает логарифм, либо при ответе на запрос. Мы же хотим избавиться от логарифмов полностью.

7.3 RMQ± 1 online. Маленьких отрезков становится мало.

Давайте для начала решим упрощенную версию задачи RMQ, после чего сведем общий случай к простому.

Простая версия (RMQ±1) отличается от обычной задачи RMQ тем, что в данном нам массиве соседние элементы обязательно отличаются друг от друга на ±1. Как же это нам поможет искать минимум на маленьких подотрезках?

Давайте заметим, что позиция минимума на отрезке определяется только разностями соседних элементов и не зависит от самих значений этих элементов. К примеру, можно считать, что первый элемент отрезка равен нулю (как будто мы вычли из всех элементов на отрезке первый), и в таком массиве минимум будет стоять на той же самой позиции, что и в изначальном. Поменяется лишь его значение. Но если мы найдем позицию минимума, найти его значение не составит труда: нужно лишь обратиться к соответствующему индексу изначального массива.

Теперь заметим, что если мы превратим все блоки в последовательности разностей соседних элементов, то эти последовательности будут состоять из чисел 1 и -1, тогда если длина блока равна k, то существует всего 2^{k-1} различных последовательностей разностей соседних элементов. После чего мы можем заранее предпосчитать для каждой такой последовательности минимум на каждом подотрезке. На это уйдет $O(2^{k-1} \cdot k^2)$ времени. И затем просто для каждого блока в массиве определить, к какому типу он относится, и пользоваться предпосчитанными значениями позиций минимумов для всех маленьких подотрезков для ответа на запросы в будущем.

Остается лишь выбрать число k так, чтобы $2^{k-1} \cdot k^2$ было не больше n, и при этом разреженная таблица на блоках длины k строилась все еще за $O(n)$. Давайте возьмем $k = \frac{\log n}{2}$. В таком случае

$$2^{k-1} \cdot k^2 \leqslant 2^k \cdot k^2 = 2^{\frac{\log n}{2}} \cdot \log^2 n = n^{\frac{1}{2}} \cdot \log^2 n = \sqrt{n} \log^2 n$$

Это, безусловно, асимптотически меньше, чем n.

С точки зрения реализации можно представлять любой блок длины $k-1$ из ±1 как битовую маску длины $k-1$ ($+1$ — единичный бит, -1 — нулевой бит), тогда после начального предпосчета можно будет для подотрезков изначального массива находить такую же маску и легко определять, к какому типу относится текущий блок. Предпосчет же будет просто храниться в трехмерном массиве размера $2^{k-1} \times k \times k$. Кроме того, нам больше не надо хранить отдельно минимумы на префиксах и суффиксах блоков, потому что теперь у нас есть минимумы на вообще всех подотрезках внутри блоков. Также можно в предпосчете хранить все таки не позицию, а значение минимума на соответствующем подотрезке при условии, что первый элемент равен нулю, если вам нужно только значение минимума, а не его позицию. Тогда при ответе на запрос надо будет просто прибавить к ответу элемент на левой границе отрезка.

Реализация доступна по ссылке.

Наиболее эффективная реализация доступна по ссылке.

7.4 Сведение LCA к RMQ±1.

RMQ±1 — весьма специфичная задача, однако, вероятно, вы уже с ней встречались. Один из алгоритмов поиска LCA в дереве сводит LCA к RMQ при помощи выписывания эйлерова обхода дерева. После этого обычно используется разреженная таблица для поиска минимума на отрезке. Однако можно заметить, что в эйлеровом обходе соседние вершины соединены ребром, поэтому разность их глубин в точности равна ±1, так что на самом деле LCA сводится к задаче RMQ±1, поэтому мы научились решать задачу LCA online за $O(n+q)$.

Реализация доступна по ссылке.

7.5 Сведение RMQ к LCA.

А теперь наступает неожиданный поворот. Мы сведем общий случай задачи RMQ к задаче LCA. А задачу LCA мы уже умеем сводить к задаче RMQ±1, которую мы умеем решать за $O(n+q)$. То есть такой странной последовательностью сведений мы научимся решать общий случай задачи RMQ за $O(n+q)$.

Само сведение тоже может показаться весьма странным и удивительным. Пускай у нас есть массив A. Давайте построим декартово дерево, в котором ключами будут числа от 0 до $n-1$, а приоритетами будут элементы массива A (в корне находится минимальный элемент). То есть построим декартово дерево на парах (i, A_i). В общем случае построить декартово дерево за $O(n)$ невозможно, потому что это бы решало задачу сортировки массива за $O(n)$, но в нашем случае ключи — это числа от 0 до $n-1$, так что они уже отсортированы. В случае отсортированных ключей декартово дерево можно без труда построить за $O(n)$. Алгоритм мы обсудим позже.

Теперь остается только заметить, что минимум на отрезке от l до r как раз таки соответствует LCA вершин с ключами l и r в нашем декартовом дереве. Действительно, LCA находится между левым и правым поддеревом по x координате, значит, ключ LCA лежит между l и r. С другой стороны, потенциал LCA является минимальным в его поддереве, а это поддерево содержит в себе весь подотрезок от l до r, потому что оно содержит его границы. Так что это число лежит на нужном отрезке, а также является минимумом на нем. Что и требовалось доказать.

Осталось только научиться строить декартово дерево за $O(n)$ при условии, что ключи отсортированы. Давайте последовательно добавлять элементы в дерево по возрастанию ключей. Новый ключ будет обязательно самой правой вершиной дерева. Осталось только понять, куда его подвесить, и как перестроить дерево. Давайте посмотрим на предыдущую вершину, которую мы добавили. Если потенциал новой вершины меньше или равен, чем потенциал предыдущей, то ее нужно просто подвесить к предыдущей вершине справа (так как у нее точно нет еще правого сына). Если же потенциал новой вершины меньше потенциала предыдущей вершины, нужно идти от нее наверх в сторону корня, пока мы не встретим первую вершину, у которой потенциал будет уже больше или равен потенциала новой вершины. Пускай мы добавляем в дерево вершину v, у вершины u потенциал меньше, чем у v, а у отца p вершины u потенциал уже не меньше потенциала v. Тогда теперь правым сыном вершины p будет являться вершина v, а вершину u надо подвесить левым сыном вершины v, то есть вершина v влезла внутрь ребра $p \to u$. Если же такой вершины p не существует, то есть у всех вершин потенциал меньше, чем у v, то v просто станет новым корнем, а старый корень мы подвесим к ней как левого сына.

Весь этот процесс можно представлять как стек. Давайте хранить правую ветку декартова дерева в стеке. Этот стек очень похож на стек минимумов. Тогда когда пришла новая вершина, мы удаляем элементы из стека, пока их потенциалы меньше потенциала новой вершины, а потом добавляем новую вершину в стек, производя константное количество изменений в дереве. Так как каждая вершина добавится в стек (а значит, и удалится) ровно один раз, асимптотика алгоритма — $O(n)$.

> **Замечание 7.5.1** Теперь если вы вернетесь к решению, использующему вариацию алгоритма Тарьяна, вы сможете увидеть, что тот стек, который мы использовали, как раз таки и является стеком построения декартова дерева. Однако из-за устройства алгоритма Тарьяна, мы можем явно не строить никакого дерева и получить более простой алгоритм.

Реализация доступна по ссылке.

Наиболее эффективная реализация доступна по ссылке.

7.6 Упрощенная битовая вариация алгоритма ФКБ для практики

Мы добились теоретической линейности решения RMQ, однако, этот алгоритм весьма сложен в реализации из-за трех разных фаз, а также константа в асимптотике достаточно большая, так что данный алгоритм редко можно применить на практике. При решении оффлайн версии задачи стоит посмотреть в сторону вариации алгоритма Тарьяна. Чаще всего ее будет достаточно.

Однако если задачу все таки необходимо решать в онлайне, есть упрощенная версия алгоритма ФКБ без сведений к LCA и обратно к RMQ$\pm$1. В остальном этот алгоритм похож на ФКБ. Мы все так же будем делить массив на блоки и строить разреженную таблицу на блоках. Остается научиться находить минимум на отрезках, лежащих внутри одного блока, альтернативным способом. Давайте добавим к нашему решению идею из offline решения. Пройдемся по массиву со стеком минимумов. Теперь нам остается решить две проблемы:

- Нужно сделать этот стек в некотором смысле персистентным, то есть чтобы мы могли обращаться к стеку минимумов на тот момент, когда мы находились в конкретном элементе.
- Нужно уметь за $O(1)$ находить ближайший справа элемент стека минимумов к левой границе отрезка запроса.

Но тут нам на помощь приходит тот факт, что нам надо искать минимумы только на отрезках длины $\leqslant \frac{\log n}{2}$. Это число не больше 32 при $n \leqslant 2^{64} \sim 10^{19}$. Безусловно, в реальной задаче n не может быть настолько большим. Тогда давайте хранить стек минимумов в весьма необычном формате. Давайте сохраним массив длины 32, в котором для каждого из последних 32 индексов массива будем хранить, лежит ли он в стеке минимумов или нет. Так как длина этого массива равна 32, можно не хранить массив, а сохранить это просто как битовую маску в одном числе типа `int`. При переходе к следующей позиции нужно умножить эту маску на два, удалить те биты, которые пропали из стека минимумов, а также добавить новый элемент. Так что мы можем сохранить такой миниатюрный стек для каждой позиции массива, на это уйдет $O(n)$ времени и памяти.

Теперь нам остается решить вторую проблему. Как за $O(1)$ по такой битовой маске находить минимум на отрезке длины $\leqslant 32$? Это можно сделать при помощи битовых операций. Если длина отрезка равна `len`, то применив к маске операцию побитового «и» с числом `(1 << len) - 1`, мы оставим только элементы стека минимумов, лежащие на отрезке запроса, после чего нужно найти самый первый из них. Это совпадает со старшим битом получившейся маски. Для нахождения старшего бита числа можно либо изначально предпосчитать динамику (насчитаем для всех чисел от 1 до 2^{16} старшие биты, после чего число длины 32 разбивается на два числа длины 16), либо воспользоваться встроенной в C++ функцией `__builtin_clz` (Count Leading Zeros), которая возвращает за $O(1)$ (с достаточно быстрой константой) количество лидирующих нулей в числе. Тогда чтобы получить индекс старшего бита числа, нужно просто вычесть значение этой функции из числа 32.

Асимптотика получившегося алгоритма — это вроде бы $O((n+q)\left\lceil\frac{\log n}{32}\right\rceil)$, но в действительности это $O(n+q)$, ведь при анализе времени работы алгоритмов мы всегда подразумеваем, что $n \leqslant 2^w$, где w — длина машинного слова (32 или 64), потому что если это не так, то мы не сможем хранить индексы входного массива как числа. Но если $n \leqslant 2^w$, то $\frac{\log n}{w} \leqslant 1$, поэтому данный алгоритм работает за $O(n+q)$.

Замечание 7.6.1 Обратите внимание на то, что теперь длины блоков — 32 вместо $\frac{\log n}{2}$, поэтому блоков стало меньше, и на построение разреженной таблицы уходит еще меньше времени. Кроме того, число 32 является степенью двойки, поэтому деление на длину блока будет происходить быстрее.

Замечание 7.6.2 Этот алгоритм можно оптимизировать по памяти. Нам необходимо хранить начальный массив длины n, массив миниатюрных стеков минимумов размера n, разреженную таблицу размера $\frac{n}{32}\log\frac{n}{32}$, а также обычный стек минимумов в процессе построения миниатюрных стеков. Изначальный массив и массив миниатюрных стеков нам необходимы, а вот от лишнего стека минимумов, использующегося в процессе построения, можно избавиться, ведь нам надо хранить лишь последние 32 элемента. Это можно делать различными способами, но самый простой, пожалуй, – это заметить, что нам на самом деле достаточно лишь миниатюрного стека. Чтобы обратиться к вершине стека, нужно найти в маске самый младший бит. Это делается аналогично нахождению старшего бита при помощи функции `__builtin_ctz` (Count Trailing Zeros).

Реализация доступна по ссылке.

Наиболее эффективная реализация доступна по ссылке.

Этот алгоритм примерно в 2 раза быстрее алгоритма ФКБ, а также, что самое главное, использует как минимум в 5 раз меньше памяти (есть реализации алгоритма ФКБ, которые работают не сильно медленнее приведенного алгоритма и тратят не сильно больше памяти, однако их реализации еще сложнее, чем оригинальный ФКБ).

Однако эффективная реализация дерева отрезков снизу[1] хоть и использует $O(\log n)$ времени на запрос, но на практике работает за такое же время (на удивление даже если менять ограничения, разность во времени работы получается крайне маленькой), а также тратит немного меньше памяти (примерно на четверть). Кроме того, его еще легче реализовать, а также оно может работать не только со static версией задачи, но и с динамической (к примеру, можно делать присвоение в точке).

Эффективная реализация дерева отрезков снизу доступна по ссылке.

7.7 Задачи для практики

- Задача на поиск максимума на отрезке.
- Также можете прорешать специально подготовленный контест на codeforces на эту тему. Если у вас нет доступа к соревнованию, нужно сначала вступить в группу.

[1] Смотрите 5

8. Дерево Ли Чао

8.1 Мотивация

Convex Hull Trick (CHT) позволяет отвечать на запросы поиска минимума (или максимума) набора линейных функций. Однако его проблемой является то, что добавляемые прямые должны быть монотонно упорядочены по углу наклона. Не всегда в задачах это выполнено. Конечно, CHT можно реализовать на двоичном дереве (std::set, к примеру) вместо стека, и в таком случае можно вставлять в произвольное место, удаляя устаревших соседей по обе стороны, однако это далеко не самый приятный алгоритм. Также, в силу того, что в CHT поддерживается стек, алгоритм амортизированный, так что какая-то одна операция добавления новой прямой может выкинуть из стека почти все старые прямые и будет работать за линейное время. Тогда если нам нужен персистентный Convex Hull Trick, мы этого сделать не сможем [1].

Чтобы побороть описанные выше две проблемы, нам может как раз помочь дерево Ли Чао. Это очень простая в реализации структура данных, которой не нужно, чтобы прямые были упорядочены по углу наклона. Кроме того, эта структура основана на обычном дереве отрезков, поэтому все операции будут работать за неамортизированное время, и это дерево можно делать персистентным, как и любое другое дерево отрезков.

Также, этой структуре не нужно уметь пересекать прямые, а только находить значение в точке, и ее можно использовать для произвольных функций, таких что любые две из них пересекаются не более, чем в одной точке. Помимо этого мы научимся производить массовые операции, так как мы имеем дело с деревом отрезков.

8.2 Идея

Будем решать задачу, в которой есть изначально пустое множество линейных функций L и поступают запросы двух видов:

[1] Есть способ сделать Convex Hull Trick неамортизированным и персистентным при помощи бинарных подъемов по стеку, либо спуска по декартову дереву в случае реализации с двоичным деревом поиска: 14

1. Добавить линейную функцию f в множество L.
2. Для данного x найти минимум $f(x)$ по всем функциям f из L.

Давайте построим дерево отрезков на x координатах. Обозначим максимальное значение x по модулю, которое нас интересует, за C. Тогда время обработки одного запроса к структуре будет равно $O(\log C)$. Так как C может быть очень большим числом, часто бывает ситуация, в которой дерево отрезков будет неявным. Если точки x бывают вещественные, можно сделать спуск в дереве до нужной точности. Тогда если все считается с точностью до ε, то асимптотика будет $O(\log(C \cdot \varepsilon^{-1}))$ на запрос.

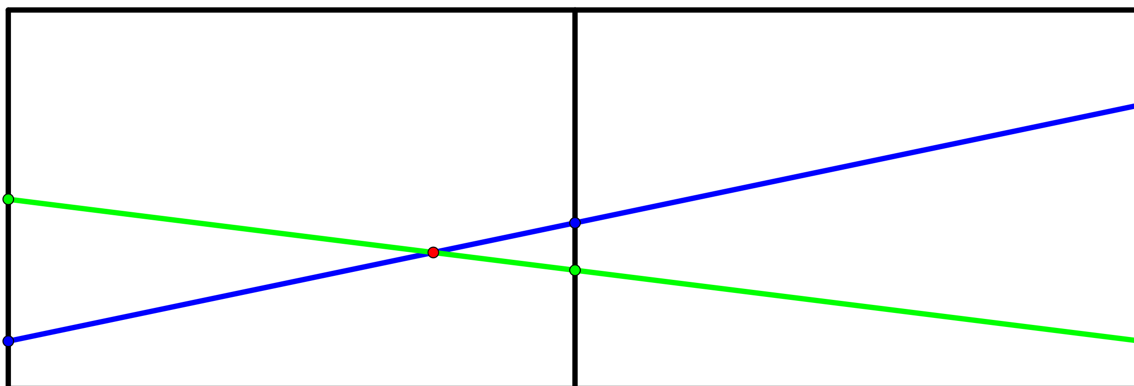
Будем в каждой вершине дерева отрезков хранить одну линейную функцию (изначально в каждой вершине хранится фиктивная функция $f(x) = +\infty$), а минимум в точке будет вычисляться как минимум по всем прямым, хранящимся в предках листа, отвечающего за эту точку.

Осталось понять, как обрабатывать запрос добавления новой прямой. Мы хотим добавить эту прямую в некоторую вершину дерева и переместить другие значения в дереве так, чтобы условие на то, что оптимальная прямая для каждой точки лежит на пути от этой точки до корня, не испортилось. Мы бы хотели положить новую прямую в корень, чтобы она лежала на пути до корня от любого листа, однако в корне, вероятно, уже лежит какая-то прямая, а мы не можем хранить сразу две прямые в вершине. Поэтому нам придется как-то решать этот конфликт.

Пускай текущая вершина отвечает за полуинтервал $[L,R)$, и у нас есть две конфликтующие функции f и g. Если одна из них меньше другой на всем полуинтервале, то нет смысла хранить бо́льшую функцию, поэтому мы можем оставить в текущей вершине меньшую и завершиться. Это можно проверить, сравнив относительный порядок значений функций f и g в точках L и R. Если в точке L одна была больше другой, а в точке R порядок поменялся, то между этими точками точно произошло пересечение.

> **Замечание 8.2.1** Ровно в этот момент мы пользуемся тем, что любые две функции должны пересекаться не более, чем в одной точке, потому что если функции пересекаются в двух точках, то они могли не изменить свой относительный порядок даже если пересекались на этом полуинтервале.

Пусть M — середина полуинтервала $[L,R)$, тогда посмотрим, в какой половине пересекаются функции. Если они пересекаются в левой половине ($[L,M)$), то в правой половине ($[M,R)$) одна из функций меньше другой во всех точках. Пусть функция f меньше g на правой половине. Тогда функция g точно не будет минимумом для точек из правой половины. Оставим функцию f в текущей вершине и протолкнем функцию g дальше в левого сына. Там опять может возникнуть конфликт, который мы будем решать аналогичным образом. В случае, если функции f и g пересекаются в правой половине, рассуждение симметрично: одна из функций точно не будет наименьшей для точек в левой половине, поэтому мы можем оставить меньшую функцию в текущей вершине и рекурсивно попытаться протолкнуть оставшуюся прямую в правое поддерево.



Можно заметить, что на самом деле прямая, которая остается в текущей вершине — это всегда прямая с меньшим значением в точке M. И точку R можно даже не подставлять в уравнения прямых. Просто если относительный порядок на прямых в точке L такой же, как в M, то в правое поддерево проталкивается прямая с бóльшим значением в точке M, а если относительный порядок разный, то эта прямая проталкивается в левое поддерево.

Можно смотреть на это иначе: в вершине дерева отрезков находится та прямая из множества, которая имеет минимальное значение в точке M из всех прямых, а оставшиеся прямые разделены по двум частям в зависимости от того, в какой из двух половин они могут быть меньше той прямой, которая хранится в текущей вершине.

Таким образом, мы будем постепенно спускаться по дереву, каждый раз запускаясь в то поддерево, в котором пересекаются конфликтующие прямые (если они не пересекаются вовсе, то мы все равно запустимся в какое-то поддерево, и в этом нет ничего страшного). Когда мы дойдем до листа, мы просто выберем меньшую функцию в этой точке и завершимся. Очевидно, что такой линейный спуск по дереву отрезков будет работать за $O(\log C)$.

С кодом можно ознакомиться по ссылке.

Замечание 8.2.2 Обратите внимание на то, что мы действительно нигде не пользовались тем, что мы работаем именно с линейными функциями, и мы их нигде не пересекали. Все, что нам было нужно, — считать значения функций в концах и середине отрезка текущей вершины, чтобы узнавать их относительный порядок. Таким образом, можно использовать эту структуру, к примеру, для набора кубических функций вида $kx^3 + b$, потому что любые две такие функции пересекаются не более, чем в одной точке.

8.3 Удаление прямых

Чтобы поддерживать не только добавление прямых в множество, но и удаление, можно воспользоваться идеей из задачи dynamic connectivity offline [2].

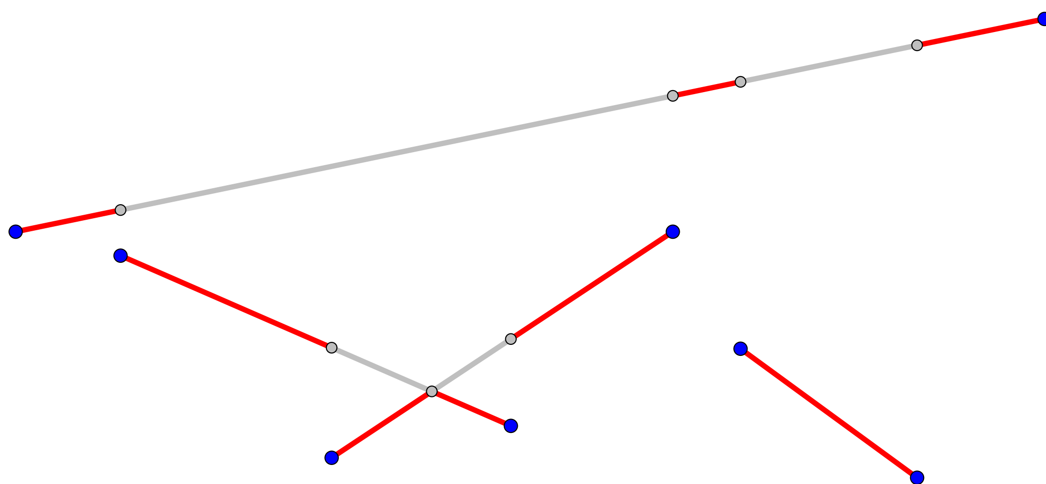
Если необходимо поддерживать структуру в online режиме, то подойдет корневая декомпозиция. Можно хранить прямые в множествах, каждое из которых имеет размер не больше K. Чтобы найти минимум, нужно перебрать все множества и запросить минимум в каждом из них. А для удаления прямой нужно перестроить с нуля всю структуру множества, из которого эта прямая удаляется. При оптимально подобранной константе K такое решение будет работать за $O\left(n\sqrt{n\log n}\right)$

[2] Подробнее можно прочитать здесь

8.4 Кусочно-линейные функции

На самом деле явно видно, что наша структура не использует дерево отрезков по полной. Мы всегда и в запросе update, и в запросе get запускаемся только в одно из поддеревьев. Давайте поймем, что дерево отрезков позволяет нам расширить спектр операций, которые может поддерживать наша структура.

Действительно, обычный запрос update в дереве отрезков применяет какую-то операцию к отрезку точек, однако мы добавляем новую прямую, которая действует сразу на все точки. Давайте научимся применять добавляемую прямую только к отрезку точек, то есть добавлять в множество не прямую, а какой-то отрезок:



Действительно, обычное дерево отрезков разбивает отрезок запроса на логарифм подотрезков. Тогда давайте запустим обычный запрос изменения к дереву отрезков, а в тот момент, когда отрезок текущей вершины лежит строго внутри отрезка запроса, мы запустим из этой вершины наше старое добавление прямой в множество, которое мы раньше запускали из корня. Таким образом, мы для каждой из логарифма вершин, в которых останавливается обычный запрос изменения для дерева отрезков, запустим нашу процедуру, работающую $O(\log C)$, и итоговая асимптотика будет $O(\log^2 C)$.

Так мы, к примеру, научились находить минимум набора кусочно-линейных функций, потому что любую кусочно-линейную функцию можно разбить на линейные отрезки и добавить их в наше множество по отдельности.

С кодом можно ознакомиться по ссылке.

8.5 Ленивые обновления

Можно пойти еще дальше и добавить новые операции к поддерживаемой структуре. Для начала давайте переформулируем решаемую задачу на языке массивов.

Пусть у нас есть массив A (A_i — это минимум значений наших линейных функций в точке i). Изначально все его элементы равны $+\infty$. Кроме того есть запросы трех видов:

1. Вставить[3] линейную функцию на полуинтервале, то есть для данных l, r, a, b присвоить $A_i = \min(A_i, a \cdot i + b)$ для всех i на полуинтервале от l до r.
2. Прибавить линейную функцию на полуинтервале, то есть для данных l, r, a, b прибавить $a \cdot i + b$ к A_i для всех i на полуинтервале от l до r.
3. Поиск оптимальной прямой в точке, то есть для данного i вывести A_i.

Комбинация первого и третьего запроса — это ровно та задача, которую мы решали раньше. Однако теперь мы можем еще менять старые прямые с помощью второй операции. При этом обратите внимание, что после второй операции «прямые» перестают быть «прямыми», потому что мы прибавляем линейную функцию к другой линейной функции, но только на отрезке, а не на всей области определения. Что же нам делать, ведь для прямых, хранящихся в некоторых вершинах дерева отрезков для некоторых точек на этом отрезке эта прямая изменится, а для других — нет.

Чтобы решить эту проблему, давайте введем операцию очищения вершины. Если в данный момент в какой-то вершине дерева хранится какая-то прямая, но она нам мешает, мы можем запустить операцию вставки этой прямой для левого и правого поддерева текущей вершины, после чего данную прямую можно будет удалить из текущей вершины. Такой процесс займет $O(\log C)$ времени.

Кроме того, будем в каждой вершине дерева хранить ленивое обновление: какую линейную функцию нужно добавить ко всем прямым на этом отрезке. Тогда операция первого типа будет работать так же, как и раньше, только перед рекурсивным запуском нужно протолкнуть ленивое обновление в детей. А операция прибавления прямой на отрезке сначала очищает все вершины, которые посетила бы стандартная операция обновления в дереве отрезков (чтобы у нас не было проблем с тем, что прямая перестает быть прямой из-за того, что к одним точкам мы прибавили значение, а к другим — нет), а затем просто оставляет ленивое обновление в логарифме вершин, на которые разбивается отрезок запроса. Такая операция работает за $O(\log^2 C)$, потому что производит очистку логарифма вершин дерева, и каждая очистка занимает $O(\log C)$ времени.

Операция третьего типа просто проталкивает ленивые обновления на пути и работает как и раньше за $O(\log C)$.

Такое модифицированное дерево Ли Чао является очень функциональным инструментом для работы с линейными функциями, и многие идейные задачи можно решить просто явным применением этой структуры. Операции можно модифицировать в зависимости от задачи. К примеру, если прибавлять на отрезке не линейную функцию, а константу, можно поддерживать не только минимум значений прямых в точке, но и на целом отрезке.

С кодом можно ознакомиться по ссылке.

8.6 Задачи для практики

- Типичная задача на convex hull trick.

[3] Немного странное слово, скорее это «добавить прямую на полуинтервале», но мы не будем использовать это слово из-за его схожести со словом «прибавить»

- Еще одна типичная задача.

Далее представлены задачи, в которых можно применить продвинутые вариации дерева Ли Чао.

- https://tlx.toki.id/problems/troc-13/F
- https://training.olinfo.it/#/task/oii_candele/statement
- https://oj.uz/problem/view/NOI20_progression
- https://atcoder.jp/contests/abc177/tasks/abc177_f
- https://codeforces.com/contest/713/problem/C

Деревья

9 Двоичные подъемы с линейной памятью
80

9. Двоичные подъемы с линейной памятью

Часто в задачах на деревья используются двоичные подъемы. Они помогают искать LCA (наименьшего общего предка), какую-то функцию на пути и так далее. Однако они занимают $O(n \log n)$ памяти. В этой главе мы рассмотрим альтернативную структуру со схожей функциональностью, занимающую линейную память.

9.1 Идея

В двоичных подъемах мы для каждой вершины храним предков на 1, 2, 4, …, 2^k вверх. Чтобы сделать структуру линейной, мы будем хранить только двух предков: непосредственного отца (`parent`) и еще какого-то одного предка (`jump`). И тогда если мы для каждой вершины еще сохраним ее глубину (`depth`), то мы сможем легко отвечать на запросы. К примеру, если нам надо найти предка текущей вершины на какой-то глубине, то каждый раз, когда мы стоим в вершине, мы будем сначала смотреть на более длинный прыжок, если он не выше нужной нам вершины, будем совершать этот прыжок, а если же он все таки выше, то просто переходить в отца. Таким способом мы гарантировано придем в нужную вершину, осталось только построить такие прыжки, чтобы этот путь занимал всегда логарифмическое количество шагов.

При этом структура будет динамическая, так же как и обычные двоичные подъемы. То есть мы можем добавлять вершины в дерево по очереди, и старые прыжки не будут пересчитываться.

Остается лишь придумать, как построить прыжки. Давайте сделаем это следующим образом: если прыжок из нашего отца (`par`) имеет такую же длину, как и прыжок из прыжка нашего отца (`jump[par]`), то мы проведем ребро в прыжок прыжка нашего отца (`jump[jump[par]]`), а иначе проведем ребро просто в нашего отца. Если вы запутались, то рекомендую осознать код:

```
1 void add_leaf(int v, int par) {
2     parent[v] = par;
3     depth[v] = depth[par] + 1;
```

```
4     if (depth[par] - depth[jump[par]] == depth[jump[par]] -
         depth[jump[jump[par]]]) {
5          jump[v] = jump[jump[par]];
6     } else {
7          jump[v] = par;
8     }
9 }
```

9.2 Доказательство

Почему же с такими прыжками нам придется совершить лишь логарифмическое количество переходов? Для понимания полезно нарисовать картинку. Заметим, что на длину прыжка из вершины `v` (а также длины прыжков из всех ее предков) влияет только ее глубина, но не структура дерева. Поэтому можно рассмотреть лишь ситуацию, в которой наше дерево является бамбуком. Давайте заметим, что длина любого прыжка равна степени двойки (если считать длину не по количеству ребер, а по количеству вершин). Действительно, для прыжка в отца длина равна 2, а любой новый прыжок — это либо прыжок в отца, либо комбинация двух одинаковых прыжков. Тогда если те прыжки имели длину, равную степени двойки, то и комбинация тоже.

Однако в отличие от обычных двоичных подъемов, в которых мы постепенно уменьшали длину прыжка, здесь все немного сложнее. Сначала длина прыжка постепенно увеличивается, а потом постепенно уменьшается. Это чем-то похоже на путешествие между городами. Сначала мы едем по маленьким улочкам, постепенно выезжая на более широкие проспекты, в конце концов выезжаем на шоссе, а в конце пути происходит симметричная ситуация: мы съезжаем с магистралей на проспекты, а с проспектов на узкие улочки.

Давайте разделим наш путь на две части: сначала длины прыжков постепенно возрастают, и мы всегда используем прыжок, потом в какой-то момент прыжок ведет в слишком высокую вершину, поэтому нам приходится использовать переход в отца, после чего мы будем иногда использовать переходы в отцов, а иногда прыжки, но длины прыжков будут не возрастать.

Докажем несколько утверждений.

Теорема 9.2.1 Прыжки не пересекаются (но один может лежать строго внутри другого).

Доказательство. Докажем это утверждение по индукции. Пусть это утверждение верно для всех предков текущей вершины, докажем для прыжка из текущей вершины. Если прыжок ведет в отца, то такой прыжок, очевидно, не может ни с кем пересекаться. Если же прыжок ведет не в отца, то он является комбинацией двух прыжков. Если какой-то прыжок пересекается с новым прыжком, то он пересекается и с одним из двух меньших прыжков. Но для них мы уже доказали по предположению индукции, что они попарно не пересекаются. ■

Теорема 9.2.2 Не может быть больше двух одинаковых прыжков подряд, то есть не может быть так, что длины прыжков из `v`, `jump[v]` и `jump[jump[v]]` совпадают.

Доказательство. Действительно, если бы они все совпадали, то прыжок из сына `jump[v]` вел бы в `jump[jump[v]]`, потому что прыжки его предков равны по длине. А

тогда этот прыжок бы пересекался с прыжком из v, что невозможно по предыдущей теореме.

Крайним случаем будет ситуация, когда длины всех трех прыжков равны единице, однако в этой ситуации из v прыжок должен вести в `jump[jump[v]]`, а не в `jump[v]`, что противоречит условию. ■

Теорема 9.2.3 Длина прыжка из v не больше длины прыжка из `jump[v]` в том случае, если `jump[v]` — это не корень. То есть если мы переходим по прыжкам, то их длины не убывают.

Доказательство. Действительно, если наш прыжок ведет в корень, то оттуда уже некуда прыгать. А иначе прыжок будет иметь не меньшую длину.

Докажем это утверждение по индукции. Пусть это утверждение верно для всех предков текущей вершины, докажем для прыжка из текущей вершины. Если этот прыжок ведет в отца, то это самый маленький возможный прыжок, и следующий точно не меньше. Если же прыжок ведет не в отца, то он был получен из двух меньших прыжков длины в два раза меньше, Тогда так как для всех предков утверждение уже верно, то прыжок из нашего прыжка точно не может быть короче нашего прыжка более, чем в два раза. Однако если он короче ровно в два раза, то мы получаем три последовательных прыжка одинаковой длины, чего не бывает по предыдущей теореме. Поэтому прыжок из нашего прыжка имеет длину не меньше, чем текущий прыжок. ■

Из этих утверждений легко понять, что первая фаза путешествия, в которой мы всегда переходим по прыжкам, работает за логарифмическое время. Действительно, длина прыжка не убывает, но при этом она не может оставаться фиксированной более, чем два раза, поэтому через $2\log n$ итераций длина прыжка станет не меньше n, и в этот момент прыжок уже точно будет выше, чем необходимая нам вершина.

На второй же фазе путешествия мы пытаемся получить точную вершину, в которую нам нужно прийти. нам не подходит прыжок длины 2^k, то есть наша вершина находится между текущей вершиной и прыжком из нее. Тогда мы переходим в нашего отца, в результате чего отрезок, на котором мы ищем, поделился на два. И если нужная вершина находится в верхней половине, то мы сделаем прыжок из нашего отца, а если в нижней, то не сделаем. В любом случае, через каждые два шага длина прыжка из текущей вершины будет уменьшаться в два раза, и таким бинпоиском мы постепенно придем в нужную вершину.

9.3 Поиск предка на глубине h

Доказательство не совсем очевидно, однако его не нужно помнить, чтобы применять эту структуру данных. Давайте рассмотрим пример ее работы для задачи LA (level ancestor), то есть поиска предка текущей вершины на глубине h. Для этого мы постепенно идем наверх, пытаясь пойти в прыжок текущей вершины, если его глубина не меньше h, а в противном случае идем в отца.

С реализацией можно ознакомиться по ссылке.

Асимптотика ответа на q запросов на дереве из n вершин будет $O(n+q\log n)$.

9.4 Поиск наименьшего общего предка

Теперь рассмотрим другую задачу, для решения которой обычно используют бинарные подъемы: LCA (least common ancestor или наименьший общий предок). Воспользуемся

следующей идеей: сначала из более глубокой вершины перейдем на глубину менее глубокой при помощи LA, а затем будем параллельно подниматься наверх из двух вершин. Обратите внимание на то, что если две вершины находятся на одной и той же глубине, то длины их прыжков совпадают, поэтому мы можем переходить по прыжку в том случае, если концы этих прыжков не равны, а в противном случае переходить в отца.

С реализацией можно ознакомиться по ссылке.

Асимптотика ответа на q запросов на дереве из n вершин будет $O(n+q\log n)$.

Также аналогично двоичным подъемам можно хранить значение какой-то функции на прыжке и при помощи этого искать значение функции на пути в дереве.

9.5 Задачи для практики

Для практики подойдут любые задачи на двоичные подъемы и LCA.

- Задача на поиск k-го предка в дереве.
- Задача на поиск наименьшего общего предка.
- Задача, в которой нужно еще некоторые знания о минимальных остовных деревьях.

Динамическое программирование

10 Оптимизация Кнута — Яо 85

11 Оптимизация разделяй-и-властвуй . . 91

12 Лямбда-оптимизация 98

13 Неравенство четырехугольника 111

14 Персистентный Convex Hull Trick . . . 120

10. Оптимизация Кнута — Яо

Оптимизация Кнута — Яо — стандартная оптимизация динамики, позволяющая решать два весьма разных типа динамики (если динамика удовлетворяет некоторому условию, о котором мы поговорим позже):

- Задача об оптимальном разбиении массива из n элементов на k подотрезков ($dp[ki][ni] = \min_{i<ni} dp[ki-1][i] + w(i, ni)$)
- Динамика по подотрезкам ($dp[l][r] = w(l, r) + \min_{l<i<r} dp[l][i] + dp[i][r]$)

Оба варианта решаются при помощи этой оптимизации за время $O(n^2)$, при том что без оптимизации они решаются за $O(n^2k)$ и $O(n^3)$ соответственно.

> **Замечание 10.0.1** Чаще эта оптимизация называется просто оптимизацией Кнута, однако Кнут придумал, как использовать эту оптимизацию в одном конкретном случае (для построения оптимального двоичного дерева поиска), и его доказательства были слишком сложными. Фрэнсис Яо значительно упростила доказательства Кнута и придумала очень простые условия, гарантирующие корректность этой оптимизации для большого класса других задач.

Воспринимать материал на какой-то абстрактной задаче может быть тяжело, поэтому давайте сразу сформулируем конкретные примеры задач.

10.1 Алгоритм

10.1.1 Задача об оптимальном разбиении массива на k подотрезков

Задача 10.1 Дан массив неотрицательных чисел a длины n. Назовем мощностью подотрезка квадрат суммы его элементов. Необходимо разбить массив a на k подотрезков таким образом, чтобы минимизировать сумму мощностей подотрезков разбиения. Иными словами, выделить такие $k-1$ чисел (границы подотрезков разбиения) i_1, i_2, ..., i_{k-1}, чтобы

$$(a_1 + a_2 + \ldots + a_{i_1-1})^2 + (a_{i_1} + a_{i_1+1} + \ldots + a_{i_2-1})^2 + \ldots + (a_{i_{k-1}} + \ldots + a_n)^2$$

было минимально.

В нашем случае $dp[ki][ni]$ — минимальная стоимость разбиения первых n элементов на ki подотрезков, а $w(i,ni)$ — мощность подотрезка $(i,ni]$, то есть $(a_{i+1}+a_{i+2}+\ldots+a_{ni})^2$. Далее везде мы будем подразумевать, что $w(i,ni)$ можно считать за $O(1)$. Для этого необходимо заранее посчитать префиксные суммы массива a и выразить мощность подотрезка через квадрат разности префиксных сумм его концов.

Базовый алгоритм решает эту задачу за $O(n^2k)$: мы перебираем количество отрезков по возрастанию, перебираем индекс ni, после чего перебираем все возможные индексы i и выбираем среди них оптимальный ответ. Сходу совершенно не ясно, где здесь есть место для улучшения.

Нашу динамику можно представить как таблицу $k\times n$, и ki-й ее слой пересчитывается через $ki-1$-й. Тогда давайте заметим, что не обязательно перебирать индексы ni по возрастанию, мы можем перебирать их в любом порядке, в котором нам захочется.

Давайте заметим, что если бы мы для каждой точки ni откуда-то узнали оптимальные точки i, через которые нужно пересчитываться, то мы бы могли просто для каждого ni обновить ответ $dp[ki][ni] := dp[ki-1][i]+w(i,ni)$. Давайте обозначим такое оптимальное i как $opt[ki][ni]$ (если есть несколько индексов i, дающих оптимальный ответ, выберем среди них наименьший). Тогда зная $opt[ki][ni]$, мы можем за $O(1)$ найти $dp[ki][ni]$.

Но как же нам найти $opt[ki][ni]$? В этом нам поможет то самое условие, о котором говорилось в начале статьи, которое мы должны наложить на динамику, чтобы оптимизация Кнута — Яо работала.

Условие монотонности точки разреза по обеим координатам: Оптимизация Кнута — Яо работает тогда и только тогда, когда $opt[ki-1][ni]\leqslant opt[ki][ni]\leqslant opt[ki][ni+1]$ для любых ki и ni.

Иными словами, при фиксированном ki точка пересчета $opt[ki][ni]$ должна не убывать по ni, то есть чем правее ni, тем правее левый конец последнего отрезка. А также при фиксированном ni точка пересчета $opt[ki][ni]$ должна не убывать по ki, то есть чем больше отрезков мы можем использовать, тем короче должен быть последний отрезок. Иными словами, в матрице opt возрастают и строки, и столбцы. Если подумать, то можно понять, что эти два условия весьма логичны в общем случае, так что часто они действительно выполняются.

О том, почему эти условия выполняются в нашей задаче, мы поговорим позже. Пока что мы просто примем это на веру. На самом деле чаще всего в задачах на оптимизации динамики вы не доказываете, что необходимые условия выполняются, а лишь интуитивно это понимаете или просто верите в это.

Как же эти условия помогут нам решить задачу? Да очень просто! Давайте перебирать количество подотрезков разбиения ki по возрастанию, а индекс ni по убыванию. Тогда в момент, когда мы хотим посчитать $opt[ki][ni]$, уже посчитаны $opt[ki-1][ni]$ и $opt[ki][ni+1]$, поэтому ответ нам нужно искать между ними.

Для большего понимания приведем код алгоритма:

```
1  for(int ki = 1; ki <= k; ki++) {
2      for(int ni = n - 1; ni >= 0; ni--) {
3          int from = opt[ki - 1][ni]; // opt[0][ni] = 0
4          int to = opt[ki][ni + 1]; // opt[ki][n] = n - 1
5          for(int i = from; i <= to; i++) {
6              if(dp[ki][ni] > dp[ki - 1][i] + w(i, ni)) {
7                  dp[ki][ni] = dp[ki - 1][i] + w(i, ni);
8                  opt[ki][ni] = i;
```

```
9                 }
10            }
11        }
12    }
```

Замечание 10.1.1 Заметьте, что так как мы высчитываем все оптимальные точки пересчета, то мы можем при необходимости использовать их для восстановления оптимального разбиения без каких-либо проблем.

Почему же этот код будет работать за $O(n^2)$? Давайте проанализируем количество итераций, которые мы делаем. Для каждой конкретной пары ki, ni мы делаем $opt[ki][ni+1] - opt[ki-1][ni] + 1$ операций (мы считаем, что $opt[0][ni] = 0$ и $opt[ki][n] = n-1$). Интуитивно почти каждое слагаемое $opt[ki][ni]$ побывает в итоговой сумме один раз с плюсом и один раз с минусом и сократится. А если формально, то итоговое время работы — это сумма времен работы по всем парам ki, ni:

$$\sum_{ki=1}^{k}\sum_{ni=0}^{n-1}(opt[ki][ni+1] - opt[ki-1][ni]+1) = \sum_{ki=1}^{k}\sum_{ni=0}^{n-1} opt[ki][ni+1] - \sum_{ki=1}^{k}\sum_{ni=0}^{n-1} opt[ki-1][ni] + \sum_{ki=1}^{k}\sum_{ni=0}^{n-1} 1 =$$

$$= \sum_{ki=1}^{k}\sum_{ni=0}^{n-1} opt[ki][ni+1] - \sum_{ki=1}^{k}\sum_{ni=0}^{n-1} opt[ki-1][ni] + kn$$

Заметим, что почти все $opt[i][j]$ побывают в этой сумме один раз с плюсом и один раз с минусом. Только с плюсом побывают те элементы, для которых $ki-1 \neq i$ ни для одного ki. Это значит, что $i = k$, то есть это последняя строчка матрицы. Кроме того только с плюсом войдут элементы $opt[i][n] = n-1$, которые формально не принадлежат нашей матрице. Только с минусом аналогично войдут элементы, для которых $ni+1 \neq j$ ни для какого ni. Это значит, что $j = 0$, то есть это первая строка матрицы. Кроме того только с минусом войдут элементы $opt[0][j] = 0$, которые опять же формально не принадлежат нашей матрице.

Если сократить все одинаковые слагаемые, наша асимптотика будет иметь такой вид:

$$kn + \sum_{ni=0}^{n-1} opt[k][ni] + \sum_{ki=1}^{k} opt[i][n] - \sum_{ki=1}^{k} opt[ki][0] - \sum_{ni=0}^{n-1} opt[0][ni] \leqslant$$

$$\leqslant kn + n\cdot n + k\cdot n - k\cdot 0 - n\cdot 0 = n\cdot(k+n+k) = O(n^2)$$

Последнее равенство верно, потому что мы разбиваем на k подотрезков, а значит, $k \leqslant n$.

Эту асимптотику можно было понять немного иначе: посмотрим отдельно на каждую диагональ (то есть пары ki, ni с фиксированной разностью). Можно легко заметить, что отрезки, по которым пробегается цикл для элементов этой диагонали, пересекаются только по концам, так что для каждой диагонали мы работаем за $O(n)$. Так как диагоналей всего $n+k-1$, то итоговая асимптотика — $O(n\cdot(n+k)) = O(n^2)$.

Замечание 10.1.2 Если вы знакомы с оптимизацией динамики разделяй-и-властвуй (11), то вы могли заметить, что она решает точно такую же задачу о разбиении на k подотрезков за время $O(nk\log n)$. При этом для разделяй-и-властвуй необходимо только условие монотонности точки разреза по ni (только условие $opt[ki][ni] \leqslant$

$opt[ki][ni+1]$).

Поэтому если в задаче применима оптимизация Кнута — Яо, то в ней применима и оптимизация разделяй-и-властвуй. Они имеют несравнимые асимптотики, и в зависимости от k одна оптимизация может быть лучше другой. Если $k < \frac{n}{\log n}$, то оптимизация разделяй-и-властвуй будет быстрее, а при $k > \frac{n}{\log n}$ быстрее будет оптимизация Кнута — Яо. Однако стоит заметить, что оптимизация Кнута — Яо сильно проще в написании (просто три вложенных цикла), а также имеет меньшую константу в асимптотике в силу своей простоты (никакой рекурсии как в разделяй-и-властвуй), так что при возможности стоит использовать именно ее, пока k не значительно меньше n.

10.1.2 Динамика по подотрезкам

Абсолютно аналогично эту же оптимизацию можно применить для задачи динамики по подотрезкам вида

$$dp[l][r] = w(l,r) + \min_{l<i<r} dp[l][i] + dp[i][r]$$

Сразу обратимся к примеру задачи.

Задача 10.2 Есть последовательность из n кучек камней. В i-й из них a_i камней. За одну операцию можно объединить две соседние кучки камней в одну, и если в них суммарно было x камней, то эта операция стоит x монет. Необходимо объединить все кучки в одну, заплатив минимальное количество монет.

Если мы обозначим стоимость объединения всех кучек с l-й по r-ю (левая граница включительно, а правая не включительно) за $dp[l][r]$, то формула пересчета динамики будет именно такой, какой нужно:

$$dp[l][r] = w(l,r) + \min_{l<i<r} dp[l][i] + dp[i][r]$$

где $w(l,r)$ — это сумма размеров кучек на полуинтервале от l до r. Ее можно легко высчитывать при помощи префиксных сумм. Ответ, соответственно, будет храниться в $dp[0][n]$.

Упражнение 10.1 Опытному читателю, который уже знаком с другими стандартными задачами на оптимизацию Кнута — Яо, остается в качестве упражнения понять, почему эта задача, а также задачи построения оптимального двоичного дерева поиска и построения оптимального префиксного кода с сохранением порядка, — это на самом деле одни и те же задачи. ■

Обозначим оптимальное i в пересчете динамики за $opt[l][r]$. Тогда утверждается, что опять же выполняется условие монотонности точки разреза по обеим координатам, как и в предыдущей задаче. Значит, мы можем применить оптимизацию Кнута — Яо, перебирая левую границу отрезка по убыванию, а правую — по возрастанию (чтобы то, через что мы пересчитываемся, уже было посчитано).

```
1 for(int l = n - 1; l >= 0; l--) {
2     for(int r = l + 2; r <= n; r++) {
3         int from = opt[l][r - 1]; // opt[i][i + 1] = i
4         int to = opt[l + 1][r];
5         for(int i = from; i <= to; i++) {
6             if(dp[l][r] > w(l, r) + dp[l][i] + dp[i][r]) {
7                 dp[l][r] = w(l, r) + dp[l][i] + dp[i][r];
```

```
8                 opt[l][r] = i;
9             }
10        }
11    }
12 }
```

Так как этот код абсолютно аналогичен коду из предыдущей задачи (параметр l аналогичен параметру ni, а параметр r — параметру ki), то по тем же причинам он работает за $O(n^2)$.

Замечание 10.1.3 Иногда динамика по подотрезкам может отличаться в своей формулировке. Формула может выглядеть так:

$$dp[l][r] = \min_{l<i<r} dp[l][i] + dp[i][r] + w(l,i) + w(i,r)$$

Однако легко заметить, что эта динамика отличается лишь тем, что мы прибавляем стоимость подотрезка немного позже. Итоговое значение такой динамики изменится просто на $w(0,n) - \sum_i w(i,i+1)$, а оптимальное разбиение останется тем же самым.

10.2 Достаточное условие для монотонности точки разреза

Как мы уже поняли, необходимыми и достаточными условиями для работы оптимизации Кнута — Яо являются условия монотонности точки разреза по обеим координатам, то есть $opt[ki-1][ni] \leqslant opt[ki][ni] \leqslant opt[ki][ni+1]$ для любых ki и ni. Однако практика показывает, что доказать это утверждение обычно сложно, если не знать, как к нему подступиться. В этом разделе мы познакомимся с достаточным условием на то, чтобы условие монотонности точки разреза выполнялось, которое чаще всего несложно проверить.

Достаточные условия будут немного отличаться для двух разных видов задач, которые мы рассмотрели ранее, поэтому мы посмотрим на них отдельно.

10.2.1 Достаточное условие для задачи оптимального разбиения на k подотрезков

Если наша задача имеет первый вид, то есть

$$dp[ki][ni] = \min_{i<ni} dp[ki-1][i] + w(i,ni)$$

То необходимым условием для выполнения условия монотонности точки разреза по обеим координатам является неравенство четырехугольника для функции w. Определения и доказательства можно прочитать в специальной статье про неравенство четырехугольника (13).

10.2.2 Достаточное условие для задачи динамики по подотрезкам

Если наша задача имеет второй вид, то есть

$$dp[l][r] = w(l,r) + \min_{l<i<r} dp[l][i] + dp[i][r]$$

То для монотонности точки разреза по обеим координатам необходимы уже два условия: неравенство четырехугольника для w, а также **монотонность** функции w. Определения и доказательства опять же можно прочитать в специальной статье про неравенство четырехугольника (13).

10.3 Оптимизация Кнута — Яо для трудновычислимых стоимостей подотрезков разбиения

Везде ранее мы подразумевали, что $w(l, r)$ можно вычислить за $O(1)$ для любых индексов l и r. Однако это не всегда верно. Бывают ситуации, в которых функция w устроена каким-то сложным образом. В этом случае на помощь может прийти один простой трюк. Даже если функцию w вычислить сложно, бывает нетрудно пересчитать функцию w, если подвинуть одну из границ на единицу влево или вправо. Если это возможно, то вычисление каждого нового значения w можно выразить через последовательное движение границ предыдущего посчитанного отрезка к новому. Несложно доказать, что таких движений произойдет суммарно $O(n^2)$, поэтому если передвижение любой границы на единицу в любую сторону можно выполнять за $O(1)$, то асимптотика решения не поменяется.

Подробнее про эту технику можно прочитать в аналогичном разделе в статье про оптимизацию разделяй-и-властвуй (11).

10.4 Источники

- Статья Фрэнсис Яо, являющаяся основным источником для этой статьи
- Статья, в которой представлено доказательство достаточности неравенства четырехугольника для случая оптимального разбиения на k подотрезков
- Источник информации по разным оптимизациям динамики

10.5 Задачи

- Задача C из этого контеста — стандартная задача на построение оптимального префиксного кода с сохранением порядка
- Стандартная задача (не забудьте про быстрый ввод)
- Еще одна задача

11. Оптимизация разделяй-и-властвуй

Разделяй-и-властвуй (divide-and-conquer) — стандартная оптимизация динамики, позволяющая решать задачи об оптимальном разбиении массива из n элементов на k подотрезков за $O(nk \log n)$ (если эта динамика удовлетворяет некоторому условию, о котором мы поговорим позже), в то время как наивная динамика решает эту задачу за $O(n^2 k)$. Иными словами, мы оптимизируем динамику такого вида:

$$dp[ki][ni] = \min_{i<ni} dp[ki-1][i] + w(i, ni)$$

где $dp[ki][ni]$ — оптимальная стоимость разбить первые ni элементов массива на ki подотрезков оптимальным образом, а $w(i, ni)$ — стоимость подотрезка $(i, ni]$.

Воспринимать материал на какой-то абстрактной задаче может быть тяжело, поэтому давайте сразу сформулируем конкретный пример задачи.

11.1 Алгоритм

Задача 11.1 Дан массив неотрицательных чисел a длины n. Назовем мощностью подотрезка квадрат суммы его элементов. Необходимо разбить массив a на k подотрезков таким образом, чтобы минимизировать сумму мощностей подотрезков разбиения. Иными словами, выделить такие $k-1$ чисел (границы подотрезков разбиения) i_1, i_2, ..., i_{k-1}, чтобы

$$(a_1 + a_2 + \ldots + a_{i_1 - 1})^2 + (a_{i_1} + a_{i_1+1} + \ldots + a_{i_2-1})^2 + \ldots + (a_{i_{k-1}} + \ldots + a_n)^2$$

было минимально.

В нашем случае $dp[ki][ni]$ — минимальная стоимость разбиения первых n элементов на ki подотрезков, а $w(i, ni)$ — мощность подотрезка $(i, ni]$, то есть $(a_{i+1} + a_{i+2} + \ldots + a_{ni})^2$. Далее везде мы будем подразумевать, что $w(i, ni)$ можно считать за $O(1)$. Для этого необходимо заранее посчитать префиксные суммы массива a и выразить мощность подотрезка через квадрат разности префиксных сумм его концов.

Базовый алгоритм решает эту задачу за $O(n^2k)$: мы перебираем количество отрезков по возрастанию, перебираем индекс ni, после чего перебираем все возможные индексы i и выбираем среди них оптимальный ответ. Сходу совершенно не ясно, где здесь есть место для улучшения.

Нашу динамику можно представить как таблицу $k \times n$, и ki-й ее слой пересчитывается через $ki-1$-й. Тогда давайте заметим, что не обязательно перебирать индексы ni по возрастанию, мы можем перебирать их в любом порядке, в котором нам захочется. Давайте научимся находить очередной слой нашей динамики за $O(n \log n)$, тогда на подсчет k слоев нам как раз понадобится $O(nk \log n)$ времени.

Давайте заметим, что если бы мы для каждой точки ni откуда-то узнали оптимальные точки i, через которые нужно пересчитываться, то мы бы могли просто для каждого ni обновить ответ $dp[ki][ni] := dp[ki-1][i] + w(i, ni)$. Давайте обозначим такое оптимальное i как $opt[ki][ni]$ (если есть несколько индексов i, дающих оптимальный ответ, выберем среди них наименьший). Тогда зная $opt[ki][ni]$, мы можем за $O(1)$ найти $dp[ki][ni]$.

Но как же нам найти $opt[ki][ni]$? В этом нам поможет то самое условие, которое мы должны наложить на динамику, чтобы оптимизация разделяй-и-властвуй работала, о котором говорилось в начале статьи.

Условие монотонности точки разреза: Оптимизация разделяй-и-властвуй работает тогда и только тогда, когда $opt[ki][ni] \leqslant opt[ki][ni+1]$ для любых ki и ni.

Иными словами, при фиксированном ki точка пересчета $opt[ki][ni]$ должна не убывать по ni, то есть чем правее ni, тем правее левый конец последнего отрезка. Если подумать, то можно понять, что это условие весьма логично в общем случае, так что часто оно действительно выполняется.

О том, почему это условие выполняется в нашей задаче, мы поговорим позже. Пока что мы просто примем это на веру. На самом деле чаще всего в задачах на оптимизации динамики вы не доказываете, что необходимые условия выполняются, а лишь интуитивно это понимаете или просто верите в это.

Но как же условие монотонности точки разреза поможет нам быстро найти все значения $opt[ki][ni]$? Давайте вернемся к нашему алгоритму, который вычислял значения $dp[ki][ni]$ последовательно. По умолчанию мы знаем, что $opt[ki][ni]$ — это какое-то число из интервала $[0, ni)$. Однако если мы уже посчитали $opt[ki][ni-1]$, то так как $opt[ki][ni-1] \leqslant opt[ki][ni]$, то мы можем сократить интервал поиска до $[opt[ki][ni-1], ni)$. С первого взгляда может показаться, что это уже помогает нам находить все значения $dp[ki][ni]$ за линейное время, но это не так. Представьте, что все значения $opt[ki][ni]$ равны нулю. Это никак не противоречит условию монотонности точки разреза, однако в этом случае наша оптимизация никак нам не помогает: мы все еще для каждого индекса i перебираем все индексы от 0 до ni.

> **Замечание 11.1.1** Здесь важно не путать метод разделяй-и-властвуй и метод двух указателей. Метод двух указателей работает при более сильных условиях на динамику. Ему кроме монотонности точки разреза также необходимо, чтобы при движении точки пересчета вправо ответ сначала улучшался, а потом после оптимальной точки пересчета начинал ухудшаться. В таком случае можно найти все оптимальные точки пересчета за $O(n)$. Однако в нашем случае никто не гарантирует нам, что стоимости пересчета убывают до оптимальной точки разреза и возрастают после нее. Нам лишь сказали, что для меньших индексов оптимальная точка разреза не может быть правее, чем для бо́льших.

Что же нам делать, раз наш линейный алгоритм не сработал? Давайте попробуем

найти оптимальную точку разреза для индекса $m = \left\lfloor \frac{n}{2} \right\rfloor$ ($\lfloor . \rfloor$ означает округление вниз). Тогда если мы нашли $opt[ki][m]$, то мы знаем, что для всех индексов i, меньших m, верно, что $opt[ki][i] \leqslant opt[ki][m]$, а для всех индексов i, бóльших m, верно, что $opt[ki][i] \geqslant opt[ki][m]$. Тогда мы разбили облость поиска оптимальной точки разреза для левой и правой половин массива на два множества, которые пересекаются только по $opt[ki][m]$. Тогда, если как в примере выше все $opt[ki][ni]$ равны нулю, то мы выясним, что $opt[ki][m] = 0$, и для всей левой половины массива мы значем, что оптимальная точка разреза тогда точно равна 0.

Мы нашли оптимальную точку разреза для индекса m, после чего на границы поиска оптимальной точки разреза для левой и правой половин массива появились ограничения. Тогда давайте рекурсивно запустим поиск для левой и правой половин. Там мы будем искать оптимальные точки разреза для индекса $\left\lfloor \frac{n}{4} \right\rfloor$ от 0 до $\min(opt[ki][m], \left\lfloor \frac{n}{4} \right\rfloor - 1)$, и для индекса $\left\lfloor \frac{3n}{4} \right\rfloor$ от $opt[ki][m]$ до $\left\lfloor \frac{3n}{4} \right\rfloor - 1$, и так далее.

Каждый раз нам дают некоторый отрезок $[L, R)$ индексов массива, для которых надо найти оптимальные точки разреза, а также с более высоких уровней рекурсии у нас уже есть некоторые ограничения на эти точки разреза: мы точно знаем, что они не меньше cl и меньше cr. Тогда мы выбираем индекс $M = \left\lfloor \frac{L+R}{2} \right\rfloor$, находим для него оптимальную точку разреза на интервале $[cl, \min(M, cr))$, после чего запускаемся рекурсивно для отрезка $[L, M)$, на котором нижнее ограничение — это все еще cl, а верхнее ограничение обновляется числом $opt[ki][M] + 1$ (ведь правая граница считается невключительно), потому что все индексы в левой половине отрезка меньше M; а также запускаемся рекурсивно из отрезка $[M+1, R)$, на котором верхнее ограничение остается равным cr, а нижнее ограничение становится равно $opt[ki][M]$, потому что все индексы правой половины отрезка больше M.

Для большего понимания приведем код алгоритма:

```
// find dp[ki][ni] for all ni: l <= ni < r if cl <=
   opt[ki][ni] < cr
void divide(int ki, int L, int R, int cl, int cr) {
    if (L >= R) { // empty segment
        return;
    }
    int M = (L + R) / 2;
    int opt; // optimal partition point for dp[ki][M]
    for (int i = cl; i < min(M, cr); i++) {
        if (dp[ki][M] > dp[ki - 1][i] + w(i, M)) {
            dp[ki][M] = dp[ki - 1][i] + w(i, M);
            opt = i;
        }
    }
    divide(ki, L, M, cl, opt + 1);
    divide(ki, M + 1, R, opt, cr);
}
```

Замечание 11.1.2 Заметьте, что так как мы высчитываем все оптимальные точки пересчета, то при необходимости мы можем их сохранить и использовать для восстановления оптимального разбиения без каких-либо проблем.

В конце такой рекурсивной процедуры мы найдем значения $dp[ki][ni]$ для всех индексов массива. Остается лишь понять, почему этот сложный рекурсивный процесс

разделяй-и-властвуй работает за $O(n \log n)$.

Каждый раз мы делим длину рассматриваемых отрезков в два раза, поэтому глубина рекурсии будет равна $O(\log n)$. Давайте докажем, что все вызовы с одного уровня рекурсии суммарно отработают за $O(n)$.

В одном конкретном вызове мы делаем константное количество операций, а также запускаем цикл от cl до $\min(cr, M)$. Можно считать, что один рекурсивный запуск работает за $O(cr - cl)$. На верхнем уровне рекурсии $cl = 0$ и $cr = n$, так что это $O(n)$. После чего на следующем уровне рекурсии полуинтервал $[cl, cr)$ делится на два: $[cl, opt]$ и $[opt, cr)$, и так далее. На каждом уровне рекурсии изначальный полуинтервал $[0, n)$ разбивается на несколько отрезков: $[0, i_1]$, $[i_1, i_2]$, $[i_2, i_3]$, $\ldots$ $[i_j, n)$. Соседние отрезки пересекаются по одному элементу, так что на одном уровне рекурсии каждый индекс i будет использован для пересчета максимум два раза, поэтому всего мы сделаем $O(n)$ операций на одном уровне рекурсии, что в итоге даст нам $O(n \log n)$ на все уровни. Таким образом, итоговая асимптотика алгоритма — $O(nk \log n)$, что и требовалось доказать.

> **Замечание 11.1.3** Обратите внимание на то, что мы нигде не пользовались тем, чему собственно равны $opt[ki][ni]$, нам только важно, что они не убывают. С первого взгляда может показаться, что нам нужно, чтобы opt тоже делил отрезок $[cl, cr]$ примерно пополам, но как видно из доказательства, это нигде не используется.

Последующие разделы не являются необходимыми при первом знакомстве с оптимизацией разделяй-и-властвуй. Рекомендуется прочитать их тогда, когда вы уже будете иметь некоторый опыт решения задач этим методом.

11.2 Достаточное условие для монотонности точки разреза

Как мы уже поняли, необходимым и достаточным условием для работы оптимизации разделяй-и-властвуй является условие монотонности точки разреза, то есть $opt[ki][ni] \leqslant opt[ki][ni+1]$ для любых ki и ni. Однако практика показывает, что доказать это утверждение обычно сложно, если не знать, как к нему подступиться. В этом разделе мы познакомимся с достаточным условием на то, чтобы условие монотонности точки разреза выполнялось, которое чаще всего несложно проверить.

Это условие — неравенство четырехугольника для функции w. Определения и доказательства можно прочитать в специальной статье про неравенство четырехугольника (13).

11.2.1 Лямбда-оптимизация и 1D1D — не панацея

Если вы уже читали статью про лямбда-оптимизацию (12), вы могли заметить, что если функция w удовлетворяет неравенству четырехугольника, то задачу оптимального разбиения на k подотрезков можно решить при помощи комбинациии лямбда-оптимизации и 1D1D-оптимизации за $O(n \log n \log C)$, что чаще всего лучше, чем $O(nk \log n)$ у разделяй-и-властвуй. Получается, в большинтве ситуаций разделяй-и-властвуй бесполезна? Не совсем!

Во-первых, разделяй-и-властвуй находит ответ не только для какого-то фиксированного k, но и для всех меньших, в то время как лямбда-оптимизация помогает нам найти ответ только для конкретного k. Если в задаче нужно вывести ответ для всех $ki \leqslant k$, то разделяй-и-властвуй будет работать быстрее.

Во-вторых, разделяй-и-властвуй написать чаще всего проще. Особенно в случаях, когда надо восстанавливать ответ.

В-третьих, стоит заметить, что решение лямбда-оптимизацией работает за $O(n\log n\log C)$ при условии, что функцию w можно легко вычислять за $O(1)$, однако это не всегда так! Иногда вычисление функции w — это сложно и может занимать хоть линейное время.

Но ведь разделяй-и-властвуй тоже пользуется тем, что функцию w мы вычисляем быстро, скажете вы. На самом деле это необязательно, давайте разберемся с этим в следующем разделе.

11.3 Разделяй-и-властвуй для трудновычислимых стоимостей подотрезков разбиения

Когда мы раньше говорили про стоимость подотрезка $w(i,ni)$, мы всегда подразумевали, что эту стоимость можно легко вычислить за $O(1)$, но это не всегда так. Что делать в противном случае? Рассмотрим пример задачи:

Задача 11.2 Дан массив чисел длины n. Стоимостью подотрезка этого массива назовем количество пар элементов этого подотрезка, значения которых совпадают. Необходимо разбить массив на k подотрезков, минимизировав суммарную стоимость подотрезков разбиения.

В нашем случае w — это количество пар одинаковых элементов на подотрезке. Несложно доказать, что w удовлетворяет неравенству четырехугольника, однако совершенно неясно, как находить w для произвольного подотрезка быстро. Что же делать?

Первым делом стоит потратить $O(n\log n)$ времени и сжать элементы массива (можно и $O(n)$, но это непринципиально). Теперь все элементы массива не превосходят n, но при этом какие элементы были одинаковыми, такие и остались. Теперь можно легко вычислять $w(l,r)$ за $O(r-l)$: пройдемся по отрезку, посчитаем массив cnt, в котором для каждого значения сохраним количество элементов с таким значением, после чего ответом будет сумма $\frac{cnt[c]\cdot(cnt[c]-1)}{2}$ по всем числам c. Тогда итоговая асимптотика разделяй-и-властвуй будет $O(n^2k\log n)$ вместо $O(nk\log n)$. Нельзя ли сделать как-то лучше?

На самом деле можно. Давайте заметим, что если у нас есть массив cnt и ответ для отрезка $[a,b]$, то мы можем легко пересчитать этот массив и ответ для подотрезка $[a-1,b]$. Пусть на позиции $a-1$ стоит элемент x. Тогда $w(a-1,b)=w(a,b)+cnt[x]$, а в массиве cnt нужно лишь прибавить единицу на позиции x. Давайте заметим, что для фиксированной правой границы m в процессе работы разделяй-и-властвуй мы перебираем левую границу из динапазона $[cl,cr]$. Тогда если мы изначально за $O(n)$ посчитаем $w(cr,m)$, то далее все остальные $w(i,m)$ мы сможем посчитать суммарно за $O(cr-cl)$, что как раз таки совпадает с асимптотикой, если бы мы считали w за константное время.

Посмотрим, какая в этом случае будет асимптотика: на каждом уровне динамики для каждой позиции массива мы один раз насчитываем $w(cr,m)$ за $O(n)$, а потом все остальное будет работать как и раньше суммарно за $O(n\log n)$. Поэтому итоговая асимптотика — $O(n^2k+nk\log n)=O(n^2k)$. Немного лучше, но все еще плохо. За такую же асимптотику работало бы решение динамикой без оптимизации, если бы мы воспользовались трюком с расширением отрезка. Нужно улучшать сильнее!

Давайте попробуем еще чаще переиспользовать уже посчитанные значения w. Давайте заметим, что аналогично переходу от отрезка $[a,b]$ к $[a-1,b]$, мы можем перейти и к отрезку $[a+1,b]$, а также мы аналогично можем двигать и правую границу.

Тогда пускай мы сейчас ищем оптимальный ответ для позиции m с отрезка $[l,r]$, пересчитываясь через все позиции из отрезка $[cl,cr]$. Мы перебрали все эти позиции, к примеру, по убыванию, поняли, что оптимальная среди них — это cm, и в данный момент у нас посчитан массив cnt для отрезка $[cl,m]$. После чего нам нужно рекурсивно запуститься из левой и правой половин и искать оптимальные точки пересчета для индексов $m_l = \left\lfloor \frac{l+m-1}{2} \right\rfloor$ и $m_r = \left\lfloor \frac{m+1+r}{2} \right\rfloor$ среди позиций $[cl,cm]$ и $[cm,cr]$ соответственно. Но ведь эти позиции не так далеко от текущего отрезка $[cl,m]$, для которого на данный момент посчитан массив cnt. Давайте будем двигать левую и правую границу текущего отрезка, пока он не станет равен $[cm,m_l]$, после чего мы сможем продолжить рекурсивно считать динамику для левой половины, затем, когда мы закончим рекурсивный вызов из левой половины, мы сможем подвинуть текущий отрезок в $[cr,m_r]$ и продолжить рекурсивное вычисление динамики из левой половины.

Таким образом, мы не начинаем считать массив cnt для каждой правой границы по отдельности, а переиспользуем уже вычисленные значения. В любой момент времени если нам надо посчитать w для отрезка $[a,b]$, а на данный момент у нас есть массив cnt для отрезка $[c,d]$, мы просто постепенно двигаем левую и правую границу отрезка на 1, пока не получим нужный отрезок.

Замечание 11.3.1 Обратите внимание, что если, к примеру, $a < b < c < d$, то если мы будем двигать сначала правую границу от d до b, а потом левую от c до a, то в какой-то момент у нас будет массив cnt для отрезка $[c,b]$ отрицательной длины, что может привести к каким-то ошибкам. Аналогичная ситуация может произойти, если $c < d < a < b$. Чтобы избежать такой проблемы, стоит сначала применять операции, которые расширяют отрезок, а потом уже те, которые его укорачивают. То есть сначала нужно уменьшать левую границу и увеличивать правую, пока это нужно, а потом уже увеличивать левую и уменьшать правую.

Давайте оценим время работы получившегося алгоритма. Кроме стандартных $O(nk\log n)$ на разделяй-и-властвуй, мы теперь еще тратим время на передвижение левых и правых границ текущего отрезка, на котором посчитано w. Но при этом передвижение границы на 1 занимает $O(1)$ времени, так что нам достаточно лишь посчитать количество таких перемещений.

Давайте докажем, что когда мы запустились от подотрезка $[l,r]$ и ищем для этих индексов оптимальные точки пересчета среди точек на отрезке $[cl,cr]$, на текущем уровне рекурсии мы сделаем $O((cr-cl)+(r-l))$ перемещений. Почему это так? Изначально границы отрезка стоят на $[cr,m]$, после чего мы постепенно передвигаем левую границу вплодь до cl, на что мы тратим $cr-cl$ перемещений. Затем нам нужно переместить границы в $[cm,m_l]$. При этом левая граница все еще остается на отрезке $[cl,cr]$, а правая все еще остается на отрезке $[l,r]$, поэтому количество перемещений не будет превышать $(cr-cl)+(r-l)$. После чего мы запустимся рекурсивно из левой половины массива, там мы будем как-то двигать границы, но это нас не касается, потому что эти перемещения учтутся на более глубоких уровнях рекурсии. После окончания рекурсивного вызова границы текущего отрезка w могут быть абсолютно произвольными, однако мы точно знаем, что левая граница все еще лежит на отрезке $[cl,cr]$, а правая лежит на $[l,r]$, потому что внутри рекурсии мы пытаемся пересчитывать динамику для индексов с отрезка $[l,r]$ через отрезок $[cl,cr]$, поэтому чтобы от текущего отрезка перейти к отрезку $[cr,m_r]$, нам опять понадобится не более $(cr-cl)+(r-l)$ операций перемещения. Больше перемещений на текущем уровне рекурсии не будет, так что мы доказали, что мы сделаем $O((cr-cl)+(r-l))$ операций перемещения.

Раньше на одном уровне рекурсии мы работали за $O(cr-cl)$, поэтому добавление

еще одного $O(cr - cl)$ будет все еще суммироваться в $O(nk \log n)$. Осталось лишь доказать то же самое про $O(r - l)$. Но это можно оценить точно так же! Это ведь длины отрезков нашего разделяй-и-властвуй. На каждом уровне сумма длин этих отрезков не превышает n, потому что эти отрезки не пересекаются. А так как уровней рекурсии будет $O(\log n)$, то итоговая асимптотика будет равна $O(nk \log n)$. Точно такая же, как и для w, вычисляемой за $O(1)$. С примером кода можно ознакомиться по ссылке.

> **Замечание 11.3.2** Как вы могли уже понять, никуда специально двигать границы запроса на самом деле не нужно. Просто когда мы хотим узнать стоимость какого-то подотрезка $[a, b]$, мы двигаем границы от той позиции, в которой они сейчас находятся, к нужному отрезку.

11.4 Источники

- Источник информации по разным оптимизациям динамики

11.5 Задачи

- Несложное применение идеи
- Стандартная задача на разделяй-и-властвуй (не забудьте про быстрый ввод)
- Задача изначально не на разделяй-и-властвуй, но ее можно применить
- Задача из раздела про трудновычислимую стоимость подотрезков разбиения
- Еще одна задача
- Неожиданное применение идеи разделяй-и-властвуй
- ОСТОРОЖНО!!! СПОЙЛЕРЫ К IOI!!! Сложная задача с IOI 2014

12. Лямбда-оптимизация

Лямбда-оптимизация (также известная как aliens trick, дискретный метод множителей Лагранжа, WQS Binary Search, Lagrangian Relaxation и т.д.) — очень мощная и красивая идея оптимизации динамики, которая стала широко известна после IOI 2016. Позже выяснилось, что подобная задача уже встречалась ранее (к примеру, в 2012 году на зимних петрозаводских сборах, в более ранних китайских олимпиадах и даже в одной научной статье 1992 года), однако действительно популярна эта оптимизация стала именно после 2016 года. Так как это новая и сложная в деталях тема, то вокруг нее есть множество мифов и заблуждений. В этой статье я попытался собрать исчерпывающий источник всей информации, которая известна про лямбда-оптимизацию. Если вы знаете какую-то идею, которая здесь не упоминается, напишите, пожалуйста, об этом мне в телеграме.

Кроме того, часто так случается, что лямбда-оптимизация внутри себя использует Convex Hull Trick, поэтому если вы с ним не знакомы, рекомендуется сначала изучить его перед прочтением этой статьи.

12.1 Идея на примере задачи

Давайте рассмотрим пример задачи, которую можно решить при помощи лямбда-оптимизации.

Задача 12.1 Дан массив неотрицательных чисел a длины n. Назовем мощностью подотрезка квадрат суммы его элементов. Необходимо разбить массив a на k подотрезков таким образом, чтобы минимизировать сумму мощностей подотрезков разбиения. Иными словами, выделить такие $k-1$ чисел (границы подотрезков разбиения) i_1, i_2, ..., i_{k-1}, чтобы

$$(a_1 + a_2 + \ldots + a_{i_1 - 1})^2 + (a_{i_1} + a_{i_1+1} + \ldots + a_{i_2-1})^2 + \ldots + (a_{i_{k-1}} + \ldots + a_n)^2$$

было минимально.

Собственно, лямбда-оптимизация обычно и применяется в подобных задачах, где нужно как-то разбить массив на k подотрезков оптимальным образом.

Давайте придумаем какое-нибудь решение. Построим двумерную динамику $dp[i][c]$ размера $n \times k$, которая будет равна минимальной стоимости разбиения первых i элементов массива на c подотрезков. Ответ на задачу, соответственно, будет храниться в $dp[n][k]$.

Чтобы пересчитать эту динамику, нужно перебрать позицию j, в которой будет заканчиваться предыдущий отрезок разбиения:

$$dp[i][c] = \min_{j<i} dp[j][c-1] + (a_{j+1} + a_{j+2} + \ldots + a_i)^2$$

Такую динамику в самом простом случае можно считать за $O(n^3k)$ или $O(n^2k)$, если насчитывать сумму $a_{j+1} + \ldots + a_i$ налету или же заметить, что она равна $pref[i] - pref[j]$, где $pref$ — массив префиксных сумм массива a.

После чего стандартные оптимизации динамики типа разделяй и властвуй, оптимизации Кнута — Яо или Convex Hull Trick могли бы попробовать уменьшить асимптотику этого решения до $O(nk\log n)$, $O(n^2)$ или $O(nk)$.

Давайте для примера рассмотрим, как решать эту задачу при помощи Convex Hull Trick. Это решение понадобится нам далее.

Перепишем формулу через префиксные суммы:

$$dp[i][c] = \min_{j<i} dp[j][c-1] + (pref[i] - pref[j])^2$$

Раскроем скобки:

$$dp[i][c] = \min_{j<i} dp[j][c-1] + pref[i]^2 - 2 \cdot pref[i] \cdot pref[j] + pref[j]^2$$

Вынесем за знак минимума фиксированный член $pref[i]^2$:

$$dp[i][c] = pref[i]^2 + \min_{j<i} dp[j][c-1] - 2 \cdot pref[i] \cdot pref[j] + pref[j]^2$$

И перегруппируем:

$$dp[i][c] = pref[i]^2 + \min_{j<i}(-2pref[j]) \cdot pref[i] + (dp[j][c-1] + pref[j]^2)$$

Получилось, что нужно найти минимум линейных функций вида $(-2pref[j]) \cdot x + (dp[j][c-1] + pref[j]^2)$ в точке $x = pref[i]$. Тогда получается, что эту динамику можно считать по слоям (по возрастанию c) за $O(nk\log n)$ при помощи Convex Hull Trick. Можно заметить, что так как элементы массива неотрицательны, то $pref[i]$ не убывают, поэтому можно воспользоваться методом двух указателей вместо бинпоиска внутри Convex Hull Trick и уменьшить время работы до $O(nk)$.

Однако что делать, если ограничения на n и k очень большие? К примеру, $n, k \leqslant 10^5$. Тут уже точно никак не обойтись этой динамикой, потому что ее размер слишком большой. В этот момент к нам на помощь как раз таки и приходит лямбда-оптимизация. Ее идея заключается в том, что мы избавимся от второй размерности в динамике. Тогда новая динамика $dp[i]$ будет просто минимальной стоимостью разбить первые i элементов массива на **сколько-то** подотрезков без ограничения на их количество. Ее формула пересчета будет выглядеть следующим образом:

$$dp[i] = \min_{j<i} dp[j] + (pref[i] - pref[j])^2$$

Однако очевидно, что в этом случае оптимальное разбиение разделит наш массив на n подотрезков длины 1, потому что именно в таком случае стоимость будет минимальна.

Чтобы этого избежать, мы будем **штрафовать за каждый новый подотрезок, который мы берем**. Давайте введем константу λ (лямбда), и тогда стоимость подотрезка будет равна не $(pref[i] - pref[j])^2$, а $(pref[i] - pref[j])^2 + \lambda$. Формула пересчета динамики поменяется от этого совсем не сильно:

$$dp[i] = \min_{j<i} dp[j] + (pref[i] - pref[j])^2 + \lambda$$

Теперь давайте подумаем, что будет происходить с ответом при разном выборе константы λ. Если лямбда равна нулю, то эта динамика ничем не отличается от динамики без штрафа. В оптимальном решении мы выберем разбиение на n подотрезков. Если же мы будем постепенно увеличивать лямбду, то стоимость использования большего количества отрезков будет возрастать, и мы постепенно будем использовать все меньше и меньше подотрезков. В пределе при очень больших лямбдах (к примеру, 10^{100}), стоимость использования еще одного подотрезка будет уже больше, чем вся стоимость разбиения, поэтому оптимальное разбиение будет использовать всего один подотрезок, в котором будут лежать все элементы массива.

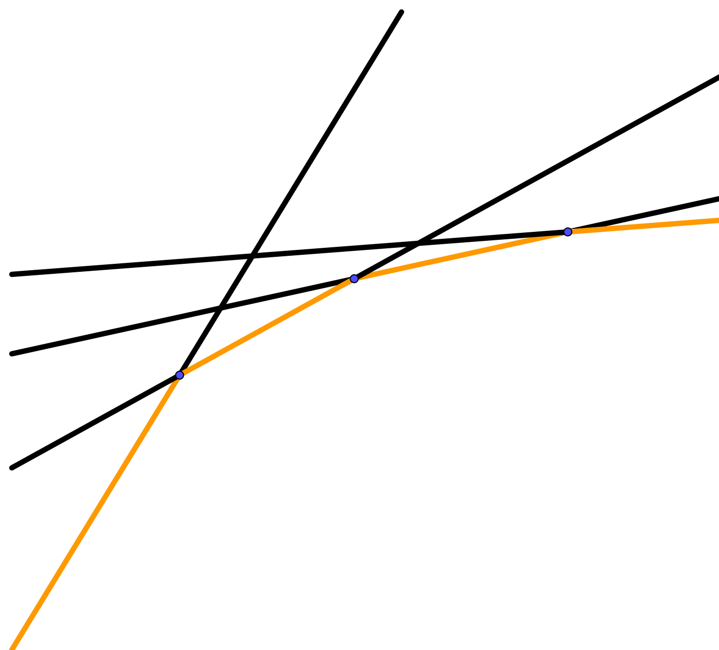
После всех этих рассуждений последняя идея уже совсем очевидна: давайте запустим бинпоиск по лямбде и найдем такую лямбду, для которой в оптимальном разбиении будет ровно k подотрезков, Тогда если ответ на задачу со штрафом равен $dp[n]$, то ответ на изначальную задачу — это $dp[n] - \lambda k$, потому что надо вычесть стоимость k подотрезков, на которые мы разбиваем наш массив. Остается лишь заметить, что формула нашей динамики с лямбдой практически такая же, как в двумерной динамике, поэтому ее можно насчитывать за $O(n \log n)$ или даже $O(n)$ при помощи Convex Hull Trick (в отличие от динамики по слоям, здесь пересчет динамики происходит через себя, поэтому прямые нужно добавлять по ходу подсчета динамики, но их угловые коэффициенты убывают, так что это можно без проблем делать), и итоговая асимптотика решения задачи будет равна $O(n \log C)$, где C — разность левой и правой границы для лямбды в бинпоиске.

Таким образом, изначальную динамику за $O(n^3 k)$ мы смогли соптимизировать до $O(n \log C)$. Невероятно!

> **Замечание 12.1.1** Если вам интересно, почему мы назвали нашу константу странной буквой λ, то дело в том, что этот алгоритм с математической точки зрения можно представлять как дискретную версию метода множителей Лагранжа, в котором по историческим причинам как раз таки используется константа λ.

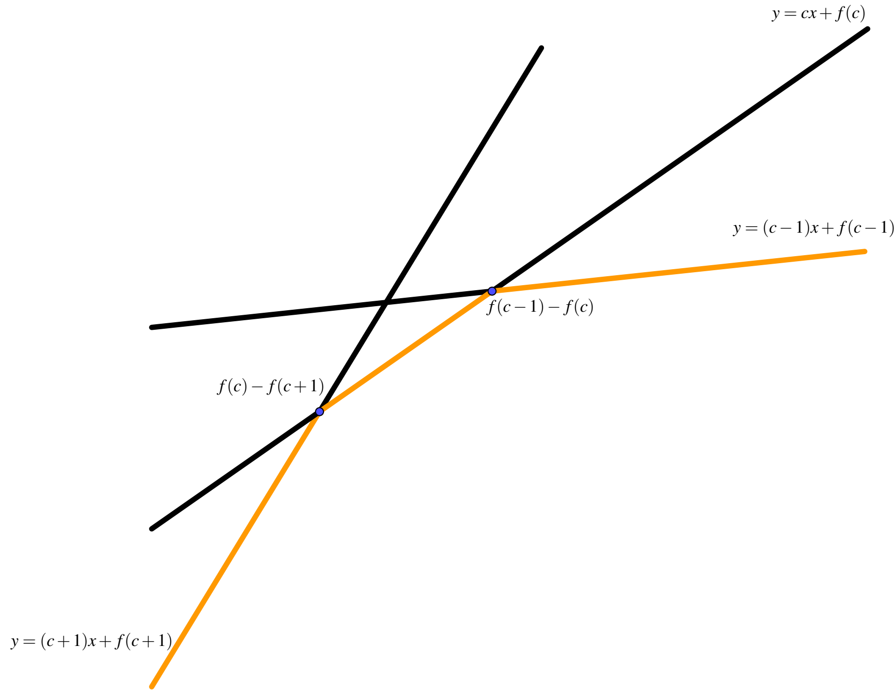
В принципе, нельзя сказать, что этот алгоритм какой-то особенно сложный, однако с ним часто возникают проблемы при реализации. Первый вопрос, который, возможно, уже пришел к вам в голову: а точно ли необходимая лямбда найдется? Не может ли быть такой ситуации, что для какой-то конкретной λ оптимальное количество подотрезков разбиения — это $k+1$, а для $\lambda + 1$ ответ уже $k-1$, и нигде не достигается ровно k подотрезков? Давайте разберемся в этом вопросе!

Определим $f(c)$ как ответ на изначальную задачу для c подотрезков (то есть $dp[n][c]$ в нашей изначальной двумерной динамике). Тогда оптимальный ответ при штрафе λ будет равен $\min_c f(c) + c \cdot \lambda$. Так как это минимум линейных функций, давайте изобразим это в виде прямых на плоскости (идея, схожая с Convex Hull Trick).



Тогда алгоритм коректен, если для каждого c есть такая точка $x = \lambda$, что в этой точке минимум достигается именно для c подотрезков. То есть, иными словами, на картинке представлен корректный Convex Hull Trick, и никакую прямую из него не надо выкинуть. В каком же случае это будет верно? Давайте посмотрим на точки пересечения соседних прямых (для c подотрезков и $c+1$ подотрезка). Тогда необходимо, чтобы эти точки пересечения возрастали.

Чему равна точка пересечения этих прямых? Это точка, в которой $f(c) + cx = f(c+1) + (c+1)x$. Это можно переписать как $x = f(c) - f(c+1)$. Мы хотим, чтобы эти точки убывали по c, то есть, иными словами, $f(c-1) - f(c) > f(c) - f(c+1)$. Возможно, вы уже поняли, что это означает. Это свойство функции называется **выпуклостью** (или выпуклостью вверх). Тогда если наша функция f удовлетворяет этому свойству, то лямбда-оптимизация будет работать. Заодно мы поняли, что **бинпоиск можно запускать только по целым числам**. Действительно, ведь точки пересечения имеют вид $f(c) - f(c+1)$, что является целым числом, ведь сами значения функции f — это целые числа (стоимости разбиений на c подотрезков).



Здесь есть одна маленькая тонкость. Пусть $f(c-1) - f(c) + 1 = f(c) - f(c+1)$, то есть точка пересечения c-й прямой с $c+1$-й на единицу больше, чем точка пересечения $c-1$-й прямой с c-й. Тогда может возникнуть такая ситуация, что при $\lambda = f(c-1) - f(c)$ мы выберем ответ, в котором будет $c-1$ отрезков, а для $\lambda = f(c) - f(c+1)$ мы уже выберем ответ с $c+1$ отрезками, и тогда c отрезков нигде не будет достигаться.

Эта проблема решается одним очень простым приемом: при подсчете динамики давайте не просто минимизировать стоимость, но и при равных стоимостях давайте минимизировать количество используемых подотрезков. От этого подсчет динамики не станет сильно сложнее. Тогда в точке $f(c+1) - f(c)$ оптимальное разбиение разобьет наш массив как раз ровно на c подотрезков.

Аналогично, лямбда оптимизация работает, если выполняется обратное условие: $f(c-1) - f(c) < f(c) - f(c+1)$. Это условие называется вогнутостью (или выпуклостью вниз). Как мы поняли, лямбда-оптимизацию можно применять, если функция является выпуклой или вогнутой. Иногда эти условия так же называются условиями **строгой** выпуклости или вогнутости. Однако на практике так получается, что почти никогда функции не удовлетворяют этим условиям. Обычно функции удовлетворяют условиям **нестрогой** выпуклости и вогнутости, которые формулируются следующим образом:

$$f(c-1) - f(c) \geqslant f(c) - f(c+1)$$

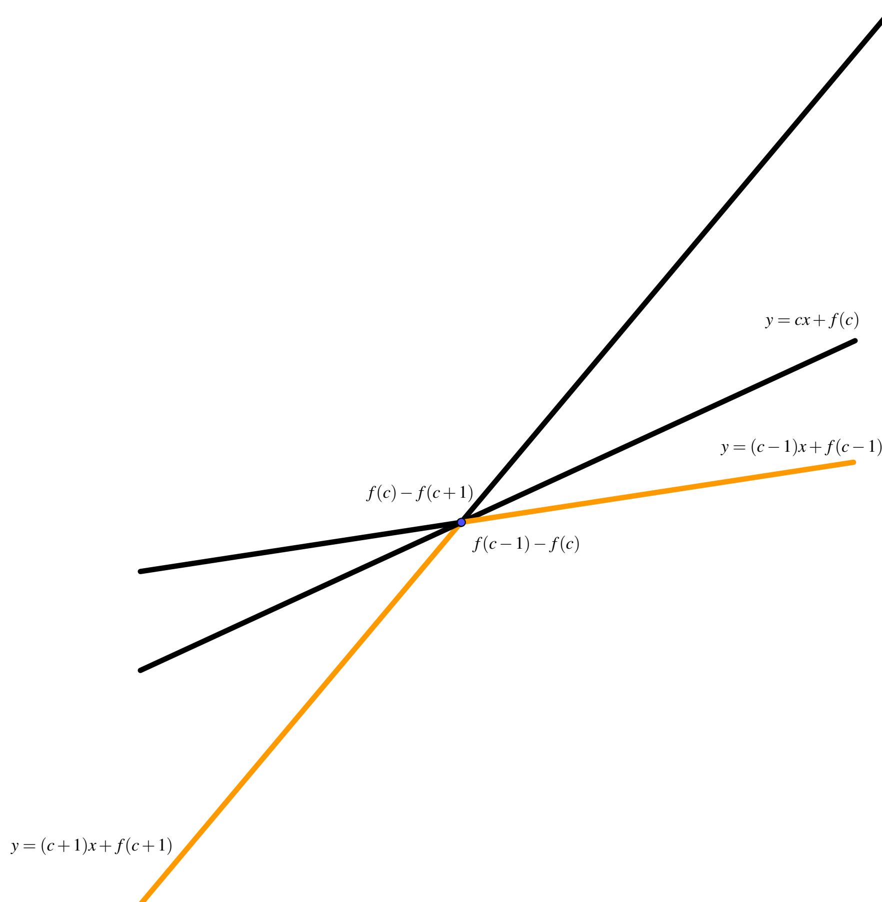
или

$$f(c-1) - f(c) \leqslant f(c) - f(c+1)$$

И на самом деле, функция f из нашей задачи тоже удовлетворяет именно условию нестрогой выпуклости. Это можно заметить на примере массива $[0,0,0]$. Для него стоимость разбиения на любое количество подотрезков равна нулю, поэтому $f(1) - f(2) = f(2) - f(3)$.

> **Замечание 12.1.2** Пока что мы не доказываем, что в нашей задаче функция f нестрого выпуклая, потому что это чаще всего совершенно не тривиальный факт. Мы докажем его (а также дадим инструмент для доказывания этого в других задачах) чуть позже. Однако обычно при написании решения этот факт просто берется на веру и не доказывается.

Что же делать, если наша функция нестрого выпуклая? В этом случае все еще существует оптимальное число λ, однако может возникнуть такая ситуация, что в одной точке пересекается сразу несколько прямых. К примеру, прямые для $c-1$, c и $c+1$. Тогда в этой точке оптимальное разбиение будет использовать $c+1$ подотрезок, а в следующей уже $c-1$ подотрезок, и так получается, что прямая для c нигде не доминирует.



Однако это не проблема! Мы все так же можем бинпоиском найти лямбду, в которой разбиение на k отрезок будет оптимальным (это наибольшая лямбда, в которой в оптимальном разбиении не менее k подотрезков). Однако если нам вернули, что в этой точке оптимально использовать c подотрезков, и стоимость равна Y, то стоимость разбиения на k подотрезков без штрафа будет попросту равна $Y - k\lambda$, ведь через точку (λ, Y) проходит прямая $f(k) + xk$. Обратите внимание, что мы вычитаем из Y именно $k\lambda$, а не $c\lambda$. $Y - c\lambda$ — это стоимость оптимального разбиения массива на c подотрезков, а нам нужно k.

12.2 Применение лямбда-оптимизации вне динамического программирования

Идею лямбда-оптимизации необязательно использовать именно для задач динамического программирования. Давайте познакомимся с примером такой задачи.
Задача 12.2 Дан граф с весами на ребрах. Необходимо найти вес минимального остовного дерева, в котором степень первой вершины равна k.

В этой задаче можно ввести число λ и прибавить его к весам всех ребер, инцидентных первой вершине, после чего найти в получившемся графе остов минимального веса. Если лямбда — большое отрицательное число, то всегда выгодно брать ребра из вершины 1, и в оптимальном остове будет максимальное их количество. Если же лямбда — большое положительное число, то брать ребра из вершины 1 очень невыгодно, и в оптимальном ответе будет только одно такое ребро.

Осталось лишь запустить бинпоиск по лямбде и найти такую лямбду, при которой в оптимальном ответе степень первой вершины будет равна k.

> **Упражнение 12.1** Дано дерево с весами на ребрах. Найдите в нем паросочетание размера k минимального веса. ■

12.3 Восстановление ответа

Мы научились находить оптимальную стоимость разбиения массива на k подотрезков при условии, что функция стоимости $f(k)$ является нестрого выпуклой. Теперь давайте поймем, как восстанавливать ответ. То есть как находить то самое оптимальное разбиение, если это спрашивается в задаче. В случае строгой выпуклости это очень просто: у нас есть конкретное λ, при котором мы разбили наш массив на k подотрезков. Необходимо просто в конце запустить еще раз подсчет динамики для оптимального λ и при подсчете поддерживать предков как в любой обычной динамике.

Проблема возникает в тот момент, когда функция f является нестрого выпуклой. В этом случае нет какого-то универсального способа восстановления ответа в лямбда-оптимизации, но мы рассмотрим два, которые обычно работают.

Вероятно, при первом знакомстве с лямбда-оптимизацией не стоит читать этот раздел, и лучше вернуться сюда, когда вы прорешаете какое-то количество задач и наберетесь опыта.

12.3.1 Хранение минимального и максимального количества подотрезков в оптимальных разбиениях

Мы уже и так храним не только значение динамики, но и минимальное количество подотрезков, на которые можно разбить массив, чтобы достигался оптимальный ответ. Давайте хранить не только минимальное количество подотрезков, но и максимальное! Назовем эти значения $l[i]$ и $r[i]$: минимальное и максимальное количество подотрезков, на которые можно разбить первые i элементов массива с данным штрафом λ, чтобы стоимость разбиения была оптимальна. Мы уже знаем, что чем больше лямбда, тем меньше отрезков нам надо использовать, и если при одной и той же лямбде можно оптимально разбить массив и на $l[i]$ подотрезков, и на $r[i]$, то можно за ту же стоимость разбить его и на любое количество подотрезков между ними.

Теперь мы можем идти с конца и постепенно восстанавливать ответ. Мы знаем, что $l[n] \leqslant k \leqslant r[n]$, потому что k подотрезков дают оптимальный ответ для данной лямбды. Теперь мы будем перебирать позиции по убыванию, пытаясь найти позицию j, через которую мы могли бы оптимально пересчитать значение динамики для n. Для этого должно, во-первых, выполняться, что $dp[n] = dp[j] + (pref[n] - pref[j])^2 + \lambda$, а

во-вторых, должно быть возможно разбить массив до позиции j оптимальным образом ровно на $k-1$ подотрезков (чтобы с добавлением еще одного весь массив разбился ровно на k подотрезков), то есть должно быть выполнено: $l[j] \leqslant k-1 \leqslant r[j]$. Когда мы нашли такое j, то мы знаем, что последний подотрезок разбиения — это $[j+1,n]$, и можем продолжать процесс поиска следующего подотрезка. Так мы за $O(n)$ сможем восстановить необходимое разбиение.

12.3.2 Второй бинпоиск для уточнения k

Пускай мы уже нашли лямбду, для которой k подотрезков являются оптимальными, но минимальное количество подотрезков в оптимальном разбиении меньше k. Тогда запустим еще один бинпоиск уже при фиксированной лямбде, который попытается сделать разбиение ровно с k подотрезками. Раньше при равной стоимости мы минимизировали количество подотрезков в разбиении, а теперь введем константу mid, такую что для индексов, меньших mid, мы будем брать большее количество подотрезков в разбиении при равной стоимости, а для индексов, не меньших mid, будем брать меньшее количество подотрезков в разбиении при равной стоимости. При $mid = 0$ мы всегда выбираем меньшее количество подотрезков и получим разбиение, в котором не больше k подотрезков, а при $mid = n$ мы всегда предпочитаем большее количество подотрезков и получим разбиение, в котором не меньше k подотрезков. Тогда мы можем запустить бинпоиск по константе mid в надежде, что найдется такая константа, для которой в оптимальном разбиении будет ровно k подотрезков. Разумеется, то, что такая константа найдется, не гарантируется.

Тогда мы сначала запускаем наше обычное решение без восстановления ответа, находим оптимальную лямбду, и уже для оптимальной лямбды запускаем бинпоиск по mid. Асимптотика этого решения — $O(n(\log C + \log n))$.

12.3.3 Комбинация двух путей

Еще про один способ восстановления ответа можно прочитать в разделе «Неравенство четырехугольника» чуть ниже (12.4.2).

12.4 Доказательство выпуклости функции f

Вероятно, при первом знакомстве с лямбда-оптимизацией не стоит читать этот раздел, и лучше вернуться сюда, когда вы прорешаете какое-то количество задач и наберетесь опыта.

Пока что мы принимали на веру, что функция $f(k)$ для нашей задачи является выпуклой или вогнутой. И чаще всего при решении вы и не будете это доказывать. Вы либо просто верите в это, либо, возможно, интуитивно понимаете, что стоимость перехода от решения с k подотрезками к решению с $k+1$ подотрезками убывает или возрастает ($f(k) - f(k+1)$ убывает или возрастает). Иными словами, можно это понимать так: если есть решения для $k+1$ и $k-1$, то из них можно переконструировать два решения для k, и тогда одно из них будет иметь вес, не больший среднего арифметического $f(k+1)$ и $f(k-1)$.

Однако если вы все такие хотите доказать выпуклость формально, то вам могут помочь методы из этого раздела. Особенно это будет полезно для авторов задач, которые хотят доказать, что их решение верно.

12.4.1 Сведение к mincost-k-flow

Пререквизиты: знание стоимостных потоков.

Стандартная задача, в которой ответ выпуклый, — это mincost-k-flow. Пусть дана сеть со стоимостями на ребрах. Определим $f(k)$ как минимальную стоимость потока величины k в этой сети. Тогда утверждается, что $f(k)$ — выпуклая функция. Действительно, давайте вспомним, как ищется поток минимального веса. Чтобы перейти от потока величины k к потоку величины $k+1$, мы находим кратчайший путь в остаточной сети и пускаем по нему единицу потока. Стоимость этого увеличения как раз равна длине кратчайшего пути. Но ведь длина кратчайшего пути не убывает при возрастании k, поэтому $f(k+1) - f(k)$ не убывает. Что и требовалось доказать.

То есть если ответ на нашу задачу можно выразить как стоимость минимального потока в какой-то сети, то функция f будет являться выпуклой, а значит, мы можем применять в этой задаче лямбда-оптимизацию. Обратите внимание, что размер сети может быть сколь угодно большой, ведь мы не собираемся реально решать нашу задачу при помощи mincost-а. Мы лишь хотим доказать, что это можно сделать. Решать же задачу мы будем лямбда-оптимизацией.

Давайте рассмотрим пример задачи, в которой можно доказать выпуклость функции f сведением к mincost-у.

Задача 12.3 Дан массив a длины n. Необходимо выбрать k непересекающихся пар соседних элементов этого массива и закрыть их доминошками так, чтобы сумма всех оставшихся чисел была максимальна.

Давайте сначала заметим, что сумма всех оставшихся чисел — это сумма всех чисел минус сумма закрытых чисел, так что нам надо минимизировать сумму закрытых чисел. Во-вторых, можно заметить, что каждая доминошка закрывает ровно одну ячейку массива на четной позиции и ровно одну на нечетной. Поэтому если мы рассмотрим двудольный граф, в одной доли которого будут четные индексы, а в другой — нечетные, и между индексами i и $i+1$ для всех i будет проведено ребро веса $a_i + a_{i+1}$, то нам нужно просто в этом двудольном графе найти паросочетание размера k минимального веса.

Но ведь поиск минимального паросочетания можно легко свести к поиску потока минимального веса. Необходимо лишь ориентировать ребра паросочетания слева направо и поставить на них пропускную способность 1, а также добавить сток и исток и провести из истока ребра нулевого веса и единичной пропускной способности во все вершины левой доли, а из всех вершин правой доли провести ребра нулевого веса и единичной пропускной способности в сток. Тогда минимальный вес потока величины k в этом графе будет как раз таки равен ответу для k доминошек для нашей изначальной задачи, а значит, мы можем решать ее при помощи лямбда-оптимизации.

12.4.2 Неравенство четырехугольника

Если наша задача заключается в разбиении массива на k подотрезков, то ее квадратичную динамику можно представить в таком виде:

$$dp[i][c] = \min_{j<i} dp[j][c-1] + w(j,i)$$

Тогда верно следующее утверждение:

Предложение 12.4.1 Если функция w удовлетворяет неравенству четырехугольника, то есть для любых $a \leqslant b \leqslant c \leqslant d$ выполнено

$$w(a,c) + w(b,d) \leqslant w(a,d) + w(b,c)$$

Тогда функция $f(c) = dp[n][c]$ является нестрого выпуклой (если неравенство в обратную сторону, то вогнутой).

Перед тем, как продолжить читать дальше, рекомендую прочитать специальную статью про неравенство четырехугольника (13). Там можно найти все необходимые определения и свойства.

Доказательство. Полностью доказывать это утверждение мы не будем, потому что это весьма объемный труд. Ознакомиться с ним можно во второй части этой статьи.

Если коротко, то доказательство состоит из последовательной проверки следующих фактов:

1. Если есть два оптимальных разбиения на отрезки P и Q, и в P один из отрезков — это $[a,b]$, а в Q один из отрезков — это $[c,d]$, и выполняется $a \leqslant c \leqslant d \leqslant b$, то эти два отрезка можно поменять на $[a,d]$ и $[c,b]$, а также поменять местами концы путей P и Q после точек d и b, и полученные пути тоже будут оптимальными. Этот факт понять весьма легко. Получившиеся пути будут тоже являться некоторыми разбиениями, однако сумма их длин будет не больше, чем сумма длин изначальных путей, по неравенству четырехугольника. Тогда если изначальные были оптимальными, то на самом деле достигается равенство, и два новых пути также являются оптимальными.
2. Для любых двух оптимальных разбиений, таких что в одном из них хотя бы на m отрезков больше, чем в другом, есть такая пара отрезков, как в прошлом факте, что если отрезок $[a,b]$ является i-м в первом разбиении, то отрезок $[c,d]$ является $i+m$-м во втором разбиении. Доказывается этот факт индукцией по обеим длинам путей.
3. И тогда совмещая два предыдущих факта, можно понять, что если есть два оптимальных пути длины k_1 и k_2, то для любого числа k между ними есть оптимальный путь длины k.
4. Из чего в конечном итоге и следует, что $f(c)$ является выпуклой.

■

Получается, что если функция стоимости отрезка в задаче удовлетворяет неравенству четырехугольника, то функция ответа является выпуклой, а значит, в этой задаче можно использовать лямбда-оптимизацию. Почему неравенство выполняется конкретно для нашей задачи, можно опять же прочитать в статье про неравенство четырехугольника (13).

Несложно проверить, что неравенство четырехугольника выполняется и для многих других задач на лямбда-оптимизацию. К примеру, накрыть точки на прямой k отрезками, минимизировав сумму квадратов их длин, или разбить массив на k подотрезков, максимизировав сумму побитового «или» подотрезков, и так далее. Опыт показывает, что большинство задач на лямбда-оптимизацию удовлетворяют этому свойству.

Применение 1D1D-оптимизации

И на самом деле это полезно не только с теоретической точки зрения! Наша формула для одномерной динамики при фиксированной лямбде выглядит следующим образом:

$$dp[i] = \min_{j<i} dp[j] + w'(j,i)$$

где $w'(j,i) = w(j,i) + \lambda$. Если w удовлетворяет неравенству четырехугольника, то и w' тоже ему удовлетворяет, потому что лямбды в неравенстве сокращаются. А одномерную динамику такого вида, в которой функция пересчета удовлетворяет неравенству четырехугольника, можно считать при помощи 1D1D-оптимизации. Поэтому на самом деле комбинация лямбда-оптимизация + 1D1D-оптимизация решает практически все задачи на лямбда-оптимизацию.

Восстановление ответа при условии неравенства четырехугольника

При доказательстве выпуклости функции f одним из промежуточных фактов мы доказали следующее утверждение:

> Лемма 12.4.2 Для любых двух оптимальных разбиений, таких что в одном из них хотя бы на m отрезков больше, чем в другом, есть такая пара вложенных отрезков $[a,b]$ и $[c,d]$, что если отрезок $[a,b]$ является i-м в первом разбиении, то отрезок $[c,d]$ является $i+m$-м во втором разбиении.

И в таком случае мы поняли, что ребра $[a,b]$ и $[c,d]$ можно заменить на $[a,d]$ и $[c,b]$, а также поменять концы двух путей, и тогда если один путь имел длину k_1, а другой — k_2, то итоговые пути будут иметь длины k_2+m и k_1-m. То есть если взять правильное m, то мы можем построить путь любой длины между k_1 и k_2.

Тогда при условии верности неравенства четырехугольника можно восстанавливать ответ следующим способом. Пусть мы уже нашли оптимальную лямбду, и для нее минимальное количество подотрезков в оптимальном разбиении — это l, а максимальное — это r. Тогда восстановим ответы для l и r (максимизируя/минимизируя количество подотрезков разбиения при равной стоимости), после чего используя эти два пути восстановим путь длины k ($l \leqslant k \leqslant r$). Необходимо просто идти по отрезкам разбиения двумя указателями и проверять, не нашли ли мы такую пару вложенных отрезков, что индекс первого в первом разбиении на $k-l$ меньше, чем индекс второго во втором разбиении. И в этом случае нужно взять префикс первого пути, суффикс второго и объединить их ребром. Вот и получился оптимальный ответ для разбиения на k подотрезков.

12.4.3 Задача линейного программирования

Для прочтения этого раздела не обязательно быть знакомым с задачей линейного программирования, но это точно сильно облегчит понимание.

В данном разделе мы рассмотрим максимально общую задачу, к которой можно сводить изначальную, чтобы ответ был выпуклым.

На самом деле нам подходят ограниченные задачи целочисленного линейного программирования, в которых переход к вещественными числам не добавляет новых базовых решений. Давайте разбираться, что все это значит.

> **Определение 12.4.1** Сформулируем задачу линейного программирования. Есть набор переменных x_1, x_2, ..., x_n. Имеется набор m условий:
>
> $a_{i,1}x_{i,1} + a_{i,2}x_{i,2} + \ldots + a_{i,j_i}x_{i,j_i} \leqslant b_i$
>
> А также линейная функция $f(x_1, x_2, \ldots, x_n) = c_1x_1 + c_2x_2 + \ldots + c_nx_n$. Необходимо найти значения переменных x_i, которые удовлетворяют всем условиям и максимизируют (или минимизируют) функцию f.
>
> Если x_i должны быть целыми, то эта задача называется задачей **целочисленного** линейного программирования.

Обозначим за b вектор, состоящий из b_i. Обозначим за $g_r(b)$ тот самый максимум функции f при данных ограничениях для задачи линейного программирования, а за $g(b)$ обозначим максимум функции f при данных ограничениях для задачи целочисленного линейного программирования.

Без труда можно понять, что $g_r(b)$ является вогнутой функцией. Давайте возьмем b_1 и b_2, а также $0 \leqslant t \leqslant 1$. Пусть x^1 максимизирует f при ограничениях b_1, а x^2 максимизирует f при ограничениях b_2. $x = tx^1 + (1-t)x^2$. Тогда раз x^1 и x^2 удовлетворяют всем ограничениям, то и x как их линейная комбинация с суммой коэффициентов 1

удовлетворяет им. А также $f(x) = tf(x^1) + (1-t)f(x^2) = tg_r(b_1) + (1-t)g_r(b_2) \leqslant g_r(tb_1 + (1-t)b_2)$, потому что f — линейная функция. Что и требовалось доказать.

Однако из того, что g_r является вогнутой, не следует, что g обязательно является вогнутой. Ведь может так получиться, что $g(b) \leqslant g_r(b)$. Однако иногда все таки верно, что $g(b) = g_r(b)$. И тогда мы как раз можем с уверенностью сказать, что функция g тоже выпуклая. А именно, это выполняется, если любое x, удовлетворяющее условиям, можно выразить как афинную комбинацию (линейную комбинацию с суммой коэффициентов, равной единице) каких-то целочисленных x^j, удовлетворяющих условиям. В таком случае мы можем записать, что $f(x)$ равно линейной комбинации $f(x^1)$ с суммой коэффициентов 1 (то есть некоторой средневзвешенной сумме), поэтому $f(x) \leqslant \max_j f(x^j) \leqslant g(b)$. Что и требовалось доказать.

Если область $\mathbb{R}^n$, где верны все условия, является ограниченной (какой-то n-мерный многогранник), то любая точка внутри него является какой-то афинной комбинацией вершин этого многогранника. Поэтому на самом деле нам достаточно проверить, что все вершины являются целочисленными. Иными словами, если некоторые неравенства обращаются в равенство таким образом, что все переменные можно однозначно восстановить.

К примеру, можно без труда доказать, что нашу задачу про сумму квадратов сумм элементов в подотрезках можно выразить как задачу линейного программирования, а потом проверить, что все вершины являются целочисленными. Введем $n + C_n^2$ переменных $x_1, x_2, \ldots, x_n, d_{i,j}$ $(1 \leqslant i < j \leqslant n)$, где x_i — номер отрезка, которому принадлежит i-й элемент, а d_{ij} говорит о том, лежат ли i-й и j-й элементы в одной группе. Мы введем следующие ограничения:

$$1 \leqslant x_1 \leqslant x_2 \leqslant \ldots \leqslant x_n \leqslant k$$

$$0 \leqslant d_{ij} \leqslant 1$$

$$1 - (x_j - x_i) \leqslant d_{ij}$$

И тогда линейная фунция $f(x_i, d_{ij}) = \sum_{i=1}^n a_i^2 + \sum_{i<j} 2d_{ij} \cdot a_i \cdot a_j$ как раз таки и будет равна ответу на нашу задачу, если мы рассматриваем целочисленную версию задачи линейного программирования. На самом деле d_{ij} может быть равно единице даже если x_i и x_j лежат в разных блоках, но от этого f только увеличится, поэтому это не точка оптимума. Теперь остается лишь проверить, что если некоторые неравенства обратились в равенства так, что все переменные восстанавливаются однозначно, то решение будет целочисленным. Это останется читателю в качестве упражнения.

Также можно заметить, что на самом деле задача поиска mincost-k-flow также является частным случаем этого метода. Можно заметить, что все ограничения в потоках выражаются в виде таких неравенств, а стоимость потока — линейная функция от них. Остается лишь вспомнить, что мы знаем, что если пропускные способности всех ребер целые, то и в оптимальном ответе по всем ребрам текут целые величины потоков.

12.5 Источники

- Хорошая статья на английском языке
- Восстановление ответа в лямбда-оптимизации
- Собрание задач и источников

- О сведении к задаче линейного программирования
- Связь с методом множителей Лагранжа
- Альтернативный способ понимания необходимости выпуклости фукнции f
- Статья 1995 года, в которой доказывается выпуклость при условии неравенства четырехугольника
- Статья 1992 года, использующая схожую идею

12.6 Задачи

- Базовый пример задачи
- Базовый пример задачи 2
- Решить задачу для бо́льшими ограничениями лямбда-оптимизацией
- Применение лямбда-оптимизации не в задаче оптимизации динамики
- Еще одно применение лямбда-оптимизации не в задаче оптимизации динамики
- USACO 2017 January Contest, Platinum, Problem 2. Building a Tall Barn
- ОСТОРОЖНО!!! СПОЙЛЕРЫ К IOI!!! Задача с IOI2016

13. Неравенство четырехугольника

13.1 Формулировки решаемых задач

Неравенство четырехугольника — это условие на стоимость подотрезков при пересчете динамики, которое используется во многих оптимизациях динамики (оптимизация Кнута — Яо (10), оптимизация разделяй-и-властвуй (11), 1D/1D-оптимизация, лямбда-оптимизация (12)). Чтобы не повторять одно и то же во всех оптимизациях, информация по неравенству четырехугольника вынесена сюда, однако обратите внимание, что нет никакого смысла читать эту статью просто так. Стоит обратить на нее внимание только в том случае, если вы изучаете какую-то конкретную оптимизацию динамики, и в ходе изучения вам понадобилось неравенство четырехугольника.

В задачах на оптимизации динамики часто есть некоторая функция стоимости $w(i, j)$, от которой зависят значения динамики. Динамика пересчитывается через предыдущие значения, но дополнительно добавляется значение этой функции w. К примеру, в задаче разбиения массива на k подотрезков оптимальным образом w как раз является стоимостью подотрезка разбиения. В динамике по подотрезкам это может быть стоимость объединения двух подотрезков в один. В одномерной динамике это может быть стоимость перехода от одной точки к следующей и т.д.

В этой статье много разных доказательств, но большинство из них имеют чисто технический характер, поэтому их изучение совершенно необязательно для понимания сути происходящего.

Примеры динамик, в которых может встретиться функция стоимости:

- Задача оптимального разбиения массива на k подотрезков:

$$dp[ki][ni] = \min_{i<ni} dp[ki-1][i] + w(i, ni)$$

- Динамика по подотрезкам:

$$dp[l][r] = w(l, r) + \min_{l<i<r} dp[l][i] + dp[i][r]$$

- Одномерная динамика:

$$dp[i] = \min_{j<i} dp[j] + w(j, i)$$

Везде далее мы будем рассматривать конкретные примеры задач, однако вы можете выбрать свою любимую задачу и пытаться доказывать условия для нее.

Пример 13.1 Задача оптимального разбиения массива на k подотрезков

Дан массив неотрицательных чисел a длины n. Назовем мощностью подотрезка квадрат суммы его элементов. Необходимо разбить массив a на k подотрезков таким образом, чтобы минимизировать сумму мощностей подотрезков разбиения. Иными словами, выделить такие $k-1$ чисел (границы подотрезков разбиения) i_1, i_2, …, i_{k-1}, чтобы

$$(a_1+a_2+\ldots+a_{i_1-1})^2+(a_{i_1}+a_{i_1+1}+\ldots+a_{i_2-1})^2+\ldots+(a_{i_{k-1}}+\ldots+a_n)^2$$

было минимально.

В нашем случае $dp[ki][ni]$ — минимальная стоимость разбиения первых n элементов на ki подотрезков, а $w(i,ni)$ — мощность подотрезка $(i,ni]$, то есть $(a_{i+1}+a_{i+2}+\ldots+a_{ni})^2$. Далее везде мы будем подразумевать, что $w(i,ni)$ можно считать за $O(1)$. Для этого необходимо заранее посчитать префиксные суммы массива a и выразить мощность подотрезка через квадрат разности префиксных сумм его концов. ■

Пример 13.2 Динамика по подотрезкам

Есть последовательность из n кучек камней. В i-й из них a_i камней. За одну операцию можно объединить две соседние кучки камней в одну, и если в них суммарно было x камней, то эта операция стоит x монет. Необходимо объединить все кучки в одну, заплатив минимальное количество монет.

Если мы обозначим стоимость объединения всех кучек с l-й по r-ю (левая граница включительно, а правая не включительно) за $dp[l][r]$, то формула пересчета динамики будет именно такой, какой нужно:

$$dp[l][r]=w(l,r)+\min_{l<i<r}dp[l][i]+dp[i][r]$$

где $w(l,r)$ — это сумма размеров кучек на полуинтервале от l до r. Ее можно легко высчитывать при помощи префиксных сумм. Ответ, соответственно, будет храниться в $dp[0][n]$. ■

Пример 13.3 Одномерная динамика

Мы будем рассматривать точно такую же задачу, как и в примере задачи оптимального разбиения массива на k подотрезков, однако у нас не будет ограничения на количество подотрезков. Вместо этого мы будем штрафовать за каждый отрезок, который мы возьмем. То есть функция w немного поменяется: $w(i,ni)=(a_{i+1}+a_{i+2}+\ldots+a_{ni})^2+c$, где c — некоторая константа. Мы добавляем эту константу, потому что несложно доказать, что без нее всегда оптимально разбить массив на n подотрезков длины 1. ■

13.2 Формулировка неравенства четырехугольника

Чтобы использовать в подобных задачах различные оптимизации динамики, мы хотим, чтобы выполнялись какие-то приятные свойства на dp (к примеру, монотонность точки разреза или выпуклость), однако чаще всего доказывать эти свойства про динамику сложно, потому что она как-то сложно определена через себя. В этом нам как раз и помогает неравенство четырехугольника. Это некоторое условие на w, из которого следуют разные условия на dp, которые нам нужны. Однако так как w чаще всего определяется как-то очень легко из условия задачи, то проверить неравенство четырехугольника для w не составляет труда. Таким образом, неравенство

четырехугольника для функции стоимости — простой способ доказывать корректность своего решения при помощи какой-то оптимизации динамики.

> **Замечание 13.2.1** Обратите внимание, что во всех контекстах, которые мы будем рассматривать, неравенство четырехугольника будет лишь достаточным условием для использования какой-то оптимизации динамики, но не необходимым. То есть если неравенство четырехугольника выполняется, то мы точно можем применить необходимую оптимизацию динамики, однако даже если неравенство четырехугольника не выполняется, это не значит, что соответствующую оптимизацию применять нельзя.

Давайте перейдем непосредственно к формулировке неравенства четырехугольника.

Неравенство четырехугольника: Говорят, что функция w удовлетворяет неравенству четырехугольника, если для любых индексов $a \leqslant b \leqslant c \leqslant d$ выполнено неравенство:

$$w(a,c) + w(b,d) \leqslant w(a,d) + w(b,c)$$

Иными словами, если один отрезок лежит строго внутри другого, то поменяв их правые границы местами, суммарная стоимость отрезков не увеличится.

> **Замечание 13.2.2** Это условие иногда еще называется **вогнутым** неравенством четырехугольника. Оно достаточно для задачи минимизации. Если же знак в неравенстве стоит в другую сторону, то это **выпуклое** неравенство четырехугольника, и оно, соответственно, является достаточным для задачи максимизации (представьте, что мы просто заменили функцию w на $-w$, тогда мы перейдем к задаче минимизации с вогнутым неравенством четырехугольника). Если выпуклость/вогнутость явно не указана, то везде далее под «неравенством четырехугольника» мы подразумеваем именно вогнутое неравенство четырехугольника. Для задач максимизации все аналогично обобщается для выпуклого неравенства четырехугольника.
>
> Также иногда эти два вида неравенств четырехугольника называют прямым и обратным.

■ **Пример 13.4** Давайте докажем неравенство четырехугольника для нашей задачи оптимального разбиения массива на k подотрезков.

В нашем случае w — это квадрат суммы элементов на отрезке. Давайте обозначим за x сумму элементов на $[a,b)$, за y сумму элементов на $[b,c)$ и за z сумму элементов на $[c,d)$. Тогда неравенство четырехугольника в нашем случае можно переписать как

$$(x+y)^2 + (y+z)^2 \leqslant (x+y+z)^2 + y^2$$

Раскрыв скобки, мы получим

$$x^2 + 2xy + y^2 + y^2 + 2yz + z^2 \leqslant x^2 + 2xy + y^2 + 2yz + z^2 + 2xz + y^2$$

Сокращая одинаковые слагаемые с обеих сторон, мы получим

$$0 \leqslant 2xz$$

Что верно, потому что оба числа x и z неотрицательны (здесь мы как раз пользуемся тем, что элементы изначального массива были неотрицательны).

■

■ **Пример 13.5** Теперь докажем неравенство четырехугольника для нашей динамики по подотрезкам.

Здесь w — это просто сумма элементов на отрезке. На самом деле в этом случае в неравенстве достигается равенство, что несложно проверить, опять же обозначив за x сумму элементов на $[a,b)$, за y сумму элементов на $[b,c)$ и за z сумму элементов на $[c,d)$. Тогда обе части неравенства равны $x+2y+z$. ■

■ **Пример 13.6** И наконец докажем неравенство четырехугольника для нашей одномерной динамики. Но ведь в этом случае функция w — это та же функция из задачи об оптимальном разбиении на k подотрезков, только к ней добавили константу. Эта константа добавится дважды к обеим частям неравенства, так что оно останется верным. ■

На самом деле из этого можно сформулировать более общее утверждение сохранения неравенства четырехугольника:

Предложение 13.2.3 **Сохранение неравенства четырехугольника при линейных преобразованиях**

Если функция w удовлетворяет неравенству четырехугольника, то функция $u(x,y) := a \cdot w(x,y) + b$ тоже удовлетворяет неравенству четырехугольника, где a — неотрицательное число, а b — любое.

Это утверждение несложно проверить, явно подставив определение u в условие неравенства четырехугольника.

> **Замечание 13.2.4** Обратите внимание, что очень важно, чтобы в предыдущем утверждении число a было неотрицательным. В противном случае знак в неравенстве поменяется на противоположный.

Предложение 13.2.5 **Сумма сохраняет неравенство четырехугольника**

Если функции w и u удовлетворяют неравенству четырехугольника, то и $v(x,y) := w(x,y) + u(x,y)$ тоже удовлетворяет неравенству четырехугольника.

Это очевидно. Достаточно лишь сложить два неравенства.

13.3 Ослабленное неравенство четырехугольника

Кроме того есть **ослабленное неравенство четырехугольника**, которое говорит, что при $a < c$ верно

$$w(a,c) + w(a+1,c+1) \leqslant w(a,c+1) + w(a+1,c)$$

Это то же самое неравенство четырехугольника, но $b = a+1$ и $d = c+1$.

Очевидно, что ослабленное неравенство четырехугольника является частным случаем обычного неравенства четырехугольника. Однако верно следующее утверждение:

Предложение 13.3.1 Если функция w удовлетворяет ослабленному неравенству четырехугольника, то она удовлетворяет и обычному неравенству четырехугольника.

Доказательство. Доказательство будет состоять из нескольких шагов, в которых мы постепенно будем доказывать неравенство четырехугольника для все более сложных четверок точек.

Сначала докажем неравенство четырехугольника, если $a = b$. В таком случае неравенство вырождается в

$$w(a,c)+w(a,d) \leqslant w(a,d)+w(a,c)$$

Слева и справа написано одно и то же, так что на самом деле достигается даже равенство. Аналогичное верно, если $c=d$.

Далее нам удобнее будет переписать наше неравенство в немного другом виде:

$w(a,c)-w(b,c) \leqslant w(a,d)-w(b,d)$

Теперь докажем это, если $b=a+1$. Доказывать будем по индукции по $d-c$. Базовый случай, когда $c=d$, мы уже доказали. Теперь сделаем переход. Мы хотим доказать, что $w(a,c)-w(a+1,c) \leqslant w(a,d)-w(a+1,d)$. Это верно, потому что

$$w(a,c)-w(a+1,c) \leqslant w(a,d-1)-w(a+1,d-1) \leqslant w(a,d)-w(a+1,d)$$

Первое неравенство верно по предположению индукции, а второе — это ослабленное неравенство четырехугольника.

Таким образом мы доказали неравенство четырехугольника при $b=a+1$. Теперь аналогично докажем для любого b. Опять перепишем неравенство в более удобном виде:

$$w(b,d)-w(b,c) \leqslant w(a,d)-w(a,c)$$

Доказывать будем по индукции по $b-a$. База $b-a=1$ уже доказана. Остался переход. Мы хотим доказать, что $w(b,d)-w(b,c) \leqslant w(a,d)-w(a,c)$. Это верно, потому что

$$w(b,d)-w(b,c) \leqslant w(a+1,d)-w(a+1,c) \leqslant w(a,d)-w(a,c)$$

Первое неравенство верно по предположению индукции, а второе — это неравенство четырехугольника для $b=a+1$, которое мы уже доказали.

Таким образом мы доказали, что из ослабленного неравенства четырехугольника следует неравенство четырехугольника для любых $a \leqslant b \leqslant c \leqslant d$. ■

Мы поняли, что вместо того, чтобы проверять стандартное неравенство четырехугольника, можно проверять ослабленное. В некоторых задачах это может упростить доказательство. К примеру, в этой:

Задача 13.1 На дороге расставлены n домов. Необходимо построить k магазинов так, чтобы суммарное расстояние от каждого дома до ближайшего магазина было минимально.

13.4 Полезные следствия неравенства четырехугольника

Обозначение 13.4.1 Обозначим за $opt(r)$ самую левую точку l, которая минимизирует $w(l,r)$, то есть $w(opt(r),r) \leqslant w(i,r)$ для любого $i<r$.

Замечание 13.4.1 Брать самую левую среди оптимальных точек l необязательно. Можно взять самую правую, и все утверждения дальше все еще будут верны.

Предложение 13.4.2 **Монотонность оптимума функции, удовлетворяющей неравенству четырехугольника**

Если w удовлетворяет неравенству четырехугольника, то $opt(r) \leqslant opt(r+1)$ для любого r. Иными словами, функция opt не убывает.

Доказательство. Предположим обратное. Пусть $opt(r) > opt(r+1)$. Тогда $w(opt(r), r+1) \geqslant w(opt(r+1), r+1)$ и $w(opt(r+1), r) > w(opt(r), r)$. Во втором неравенстве знак строгий, потому что если бы достигалось равенство, то мы выбрали бы в качестве $opt(r)$ позицию $opt(r+1)$, потому что она левее.

Теперь если просуммировать эти два неравенства, то мы получим

$$w(opt(r), r+1) + w(opt(r+1), r) > w(opt(r+1), r+1) + w(opt(r), r)$$

Что противоречит неравенству четырехугольника для $a = opt(r+1)$, $b = opt(r)$, $c = r$, $d = r+1$. ■

Предложение 13.4.3 **Неравенство четырехугольника для каждого слоя динамики оптимального разбиения массива на k подотрезков**

Для фиксированного ki обозначим $f_{ki}(l, r) := dp[ki][l] + w(l, r)$, где dp — динамика оптимального разбиения массива на k подотрезков. Тогда если w удовлетворяет неравенству четырехугольника, то f_{ki} тоже ему удовлетворяет.

Доказательство. Достаточно расписать необходимое неравенство:

$$f_{ki}(a,c) + f_{ki}(b,d) = dp[ki][a] + w(a,c) + dp[ki][b] + w(b,d) \leqslant$$

$$\leqslant dp[ki][a] + w(a,d) + dp[ki][b] + w(b,c) = f_{ki}(a,d) + f_{ki}(b,c)$$

■

Следствие 13.4.4 **Условие монотонности точки разреза для динамики оптимального разбиения массива на k подотрезков**

Если w в динамике оптимального разбиения массива на k подотрезков удовлетворяет неравенству четырехугольника, то $opt[ki][ni] \leqslant opt[ki][ni+1]$ для любых ki и ni, где $opt[ki][ni]$ — это оптимальная точка пересчета для $dp[ki][ni]$ ($dp[ki][ni] = dp[ki-1][opt[ki][ni]] + w(opt[ki][ni], ni)$). Иными словами, $opt[ki][ni]$ не убывает по ni. А значит, такую задачу можно решать при помощи оптимизации разделяй-и-властвуй (11).

Доказательство. Это является прямым следствием предыдущих двух утверждений. Во-первых, если w удовлетворяет неравенству четырехугольника, то ему удовлетворяет и f_{ki}, а $opt[ki][ni]$ — это как раз таки функция opt для f_{ki}, так что мы знаем, что она не убывает. ■

Предложение 13.4.5 **Условие монотонности точки разреза по второй координате для динамики оптимального разбиения массива на k подотрезков**

Если w в динамике оптимального разбиения массива на k подотрезков удовлетворяет неравенству четырехугольника, то $opt[ki][ni] \leqslant opt[ki+1][ni]$ для любых ki и ni. А значит, это и предыдущее условие гарантируют, что такую задачу можно решать при помощи оптимизации Кнута — Яо (10).

Доказательство. Пусть это не так. То есть $opt[ki][ni] > opt[ki+1][ni]$ для каких-то ki и ni. Посмотрим на оптимальные разбиения на ki и $ki+1$ подотрезков. Пусть оптимальное разбиение на ki подотрезков — это $[a_{ki}, a_{ki-1})$, $[a_{ki-1}, a_{ki-2})$, ..., $[a_2, a_1)$, $[a_1, ni)$, где $a_{ki} = 0$ и $a_1 = opt[ki][ni]$. Аналогично, оптимальное разбиение на $ki+1$ подотрезков — это $[b_{ki+1}, b_{ki})$, $[b_{ki}, b_{ki-1})$, ..., $[b_2, b_1)$, $[b_1, ni)$, где $b_{ki+1} = 0$ и $b_1 = opt[ki+1][ni]$.

Мы предположили, что $a_1 = opt[ki][ni] > opt[ki+1][ni] = b_1$. При этом $a_{ki} = 0 = b_{ki+1} < b_{ki}$. Давайте найдем самый большой индекс i, такой что $a_i \geqslant b_i$. То есть $a_i \geqslant b_i$ и $a_{i+1} < b_{i+1}$. Мы знаем, что i-й с конца подотрезок одного разбиения — это $[a_{i+1}, a_i)$, а i-й с конца подотрезок другого разбиения — это $[b_{i+1}, b_i)$. При этом мы знаем, что второй отрезок лежит строго внутри первого. Мы знаем, что в таком случае мы можем поменять местами правые концы отрезков и не увеличить суммарную стоимость. То есть мы можем построить новое разбиение на ki подотрезков: $[a_{ki}, a_{ki-1})$, $[a_{ki-1}, a_{ki-2})$, $\ldots$, $[a_{i+2}, a_{i+1})$, $[a_{i+1}, b_i)$, $[b_i, b_{i-1})$, $\ldots$, $[b_2, b_1)$, $[b_1, ni)$, а также мы можем построить новое разбиение на $ki+1$ подотрезков: $[b_{ki+1}, b_{ki})$, $[b_{ki}, b_{ki-1})$, $\ldots$, $[b_{i+2}, b_{i+1})$, $[b_{i+1}, a_i)$, $[a_i, a_{i-1})$, $\ldots$, $[a_2, a_1)$, $[a_1, ni)$. Заметим, что сумма стоимостей двух таких разбиений не больше, чем сумма двух оптимальных разбиений, потому что почти все отрезки остались такими, какими были, только два вложенных отрезка мы заменили на два пересекающихся, что могло только уменьшить их стоимость. Однако так как старые разбиения были оптимальны по стоимости, то новые не могут иметь меньшую стоимость. Значит, их стоимости в точности совпадают со стоимостями старых разбиений. Однако мы выбрали $opt[ki][ni]$ так, чтобы это был самый маленький индекс, через который можно оптимально пересчитаться. Но получается, что через $b_1 = opt[ki+1][ni]$ тоже можно пересчитаться оптимальным образом для ki подотрезков. Но ведь мы предполагали, что $opt[ki+1][ni] < opt[ki][ni]$. Получаем противоречие. ■

13.5 Условие монотонности по включению

Условие монотонности по включению: Говорят, что функция w монотонна (по включению), если для любых индексов $a \leqslant b \leqslant c \leqslant d$ выполнено неравенство:

$$w(b,c) \leqslant w(a,d)$$

То есть если один отрезок содержит другой, то у большего отрезка стоимость должна быть не меньше, чем у меньшего.

Это условие интуитивно еще более логично, чем неравенство четырехугольника, так что почти всегда тоже выполняется.

Среди самых первых свойств неравенства четырехугольника мы доказали, что прибавление константы и умножение на константу сохраняют свойство неравенства четырехугольника. На самом деле, если выполнено еще и условие монотонности по включению, то можно обобщить данное утверждение:

Предложение 13.5.1 **Сохранение неравенства четырехугольника и монотонности по включению при выпуклых преобразованиях:**

Если функция w удовлетворяет неравенству четырехугольника и условию монотонности по включению, то и функция $u(x,y) := f(w(x,y))$ тоже удовлетворяет неравенству четырехугольника и условию монотонности по включению, где f — неубывающая выпуклая функция.

Доказательство. То, что монотонность по включению сохраняется, — очевидно, потому что функция f не убывает.

Теперь докажем, что сохраняется неравенство четырехугольника. Мы знаем, что неравенство четырехугольника выполнено для w, то есть $w(a,c) + w(b,d) \leqslant w(b,c) + w(a,d)$. Кроме того мы знаем, что выполнено условие монотонности по включению, поэтому $w(b,c) \leqslant w(a,c), w(b,d) \leqslant w(a,d)$. Для простоты обозначим $x := w(a,c)$, $y := w(b,d)$, $z := w(b,c)$, $t := w(a,d)$. Тогда мы знаем, что $x + y \leqslant z + t$ и $z \leqslant x, y \leqslant t$. Мы же

хотим доказать, что $f(x)+f(y) \leqslant f(z)+f(t)$. При этом мы знаем, что функция f не убывает, поэтому, $f(z) \leqslant f(x), f(y) \leqslant f(t)$. Осталось заметить, что

$$f(x)-f(z) \leqslant f(t+z-y)-f(z) \leqslant f(t)-f(y)$$

Первое неравенство верно, потому что $x \leqslant t+z-y$, и при этом функция f не убывает. Второе неравенство верно из-за свойств выпуклости. Давайте обозначим $g_\alpha(x) := f(x+\alpha)-f(x)$ для какой-то неотрицательной константы α. Тогда мы знаем, что если функция f выпуклая, то g_α не убывает. Если теперь переписать второе неравенство через функцию g, то оно становится очевидным:

$$g_{t-y}(z) \leqslant g_{t-y}(y)$$

Это верно, потому что $z \leqslant y$. Что и требовалось доказать. ■

■ **Пример 13.7** К примеру, из этого следует, что раз функция, равная сумме на отрезке, удовлетворяет неравенству четырехугольника, то и квадрат суммы элементов на отрезке тоже удовлетворяет неравенству четырехугольника. ■

Предложение 13.5.2 **Неравенство четырехугольника для динамики по подотрезкам**

Если w из динамики по подотрезкам удовлетворяет неравенству четырехугольника и условию монотонности по включению, то $dp[l][r]$ тоже удовлетворяет неравенству четырехугольника.

Доказательство. Мы хотим доказать, что $dp[a][d]+dp[b][c] \geqslant dp[a][c]+dp[b][d]$. Если $a=b$ или $c=d$, то легко проверить, что это неравенство обращается в равенство. Далее будем считать, что $a<b \leqslant c<d$.

Будем доказывать это по индукции по $d-a$. База: $d-a=0$ уже доказана, потому что в этом случае $a=b$.

Теперь совершим индукционный переход. Посмотрим на $e=opt[a][d]$, то есть точку, через которую мы пересчитали значение $dp[a][d]$. Тогда $dp[a][d]=dp[a][e]+dp[e][d]+w(a,d)$. Есть три варианта, где эта точка расположена: на $(a,b]$, на (b,c) или на $[c,d)$. Мы разберем первый и второй случаи, а третий симметричен первому.

Пусть $a<e \leqslant b$. Тогда

$$dp[a][d]+dp[b][c]=dp[a][e]+dp[e][d]+w(a,d)+dp[b][c] \geqslant dp[a][e]+dp[e][c]+w(a,d)+dp[b][d] \geqslant$$

$$\geqslant dp[a][e]+dp[e][c]+w(a,c)+dp[b][d] \geqslant dp[a][c]+dp[b][d]$$

Первое неравенство верно по предположению индукции для точек e, b, c и d. Второе неравенство верно по монотонности w по включению. Третье неравенство верно, потому что $dp[a][e]+dp[e][c]+w(a,c)$ — это какой-то способ пересчитать динамику для подотрезка $[a,c)$, а $dp[a][c]$ — это оптимальный такой пересчет. Тем самым мы доказали необходимое неравенство четырехугольника.

Теперь рассмотрим второй вариант: $b<e<c$. В этом случае мы еще рассмотрим точку $f=opt[b][c]$. Не умаляя общности можно считать, что $f \leqslant e$. Другой случай будет симметричен. Тогда мы можем написать цепочку неравенств:

$$dp[a][d]+dp[b][c]=dp[a][e]+dp[e][d]+w(a,d)+dp[b][f]+dp[f][c]+w(b,c) \geqslant$$

$$\geqslant dp[a][f]+dp[e][d]+w(a,d)+dp[b][e]+dp[f][c]+w(b,c)\geqslant$$
$$\geqslant dp[a][f]+dp[e][d]+w(a,c)+dp[b][e]+dp[f][c]+w(b,d)=$$

$$=(dp[a][f]+dp[f][c]+w(a,c))+(dp[b][e]+dp[e][d]+w(b,d))\geqslant dp[a][c]+dp[b][d]$$

Первое неравенство верно по предположению индукции для точек a, b, f и e. Чтобы получить второе неравенство, мы просто применяем неравенство четырехугольника для w. И наконец третье неравенство использует то, что с левой стороны написаны какие-то способы пересчитать динамику для подотрезков $[a,c)$ и $[b,d)$, а $dp[a][c]$ и $dp[b][d]$ — это оптимальные такие пересчеты. Тем самым мы доказали необходимое неравенство четырехугольника. ■

Замечание 13.5.3 Обратите внимание, что монотонностью w по включению мы пользовались только в первом случае, но от нее нельзя избавиться.

Предложение 13.5.4 **Условие монотонности точки разреза по обеим координатам для динамики по подотрезкам**

Если w из динамики по подотрезкам удовлетворяет неравенству четырехугольника и условию монотонности по включению, то $opt[l-1][r]\leqslant opt[l][r]\leqslant opt[l][r+1]$, где $opt[l][r]$ — оптимальная точка пересчета для динамики (то есть $dp[l][r]=dp[l][opt[l][r]]+dp[opt[l][r]][r]+w(l,r)$). А значит, такую задачу можно решать при помощи оптимизации Кнута — Яо (10).

Доказательство. По предыдущему утверждению $dp[l][r]$ удовлетворяет неравенству четырехугольника. А когда мы доказывали условия монотонности точки разреза по каждой из координат для задачи оптимального разбиения на k подотрезков, мы на самом деле пользовались как раз только тем, что динамика удовлетворяет неравенству четырехугольника, так что для динамики по подотрезкам доказательства аналогичны. ■

13.6 Источники

- Хорошая статья на codeforces про значение неравенства четырехугольника в оптимизациях динамики
- Статья Фрэнсис Яо, доказывающая условия для динамики по подотрезкам
- Статья, в которой представлено доказательство достаточности неравенства четырехугольника для случая оптимального разбиения на k подотрезков
- Источник информации по разным оптимизациям динамики

14. Персистентный Convex Hull Trick

Во многих задачах на динамическое программирование используется «Convex Hull Trick» (CHT), то есть способ быстрого пересчета динамики как максимума или минимума линейных функций. Однако минусом этой техники является ее амортизированность: хранится стек прямых из выпуклой оболочки, и при добавлении новой прямой удаляются все бесполезные прямые с вершины стека. Одна конкретная такая операция может занимать $O(n)$ времени. В результате чего обычный convex hull trick нельзя сделать персистентным или использовать откаты. Для решения этой проблемы обычно используют дерево Ли Чао [1], которое является неамортизированной альтернативой CHT. В этой главе мы рассмотрим, как легко можно сделать персистентным сам convex hull trick.

Нам нужно научиться за неамортизированное время удалять большое количество элементов с вершины стека (в принципе, предложенный алгоритм подходит не только конкретно для CHT). При помощи бинпоиска мы могли бы найти последнюю прямую на стеке, которую нужно оставить, но не понятно, как удалить все ненужные прямые быстро. Однако мы все равно хотим сделать наш стек персистентным, поэтому не будем ничего удалять, а создадим новую ветку. То есть вместо стека у нас будет дерево, и если мы находимся в какой-то вершине, то текущий стек — это все элементы на пути до корня. Это сразу решает и проблему амортизированности, и проблему персистентности. Однако появляется новая проблема — и при поиске последней прямой, которую нужно оставить, и при поиске минимума линейных функций в точке, мы используем бинпоиск по стеку. В дереве мы этого сделать, к сожалению, не можем.

Для этого давайте в дереве хранить двоичные подъемы. Тогда вместо бинпоиска можно использовать подъем по дереву при помощи двоичных подъемов, который используется при поиске LA и LCA. Так как после каждой итерации добавляется ровно одна вершина в дерево как лист, то двоичные подъемы можно пересчитывать на лету для новой вершины. Асимптотика получившегося алгоритма — $O((n+q)\log n)$ на n запросов добавления прямых и q запросов получения минимума в точке. Используемая

[1] 8

память — $O(n \log n)$. При помощи двоичных подъемов с линейной памятью [2] можно уменьшить потребляемую память до $O(n)$.

Упражнение 14.1 Дано подвешенное дерево. p_v — предок вершины v ($p_v < v$). В каждой его вершине v находится какая-то линейная функция $v \cdot x + b_v$ и число a_v. Для каждой вершины дерева необходимо найти минимум по линейным функциям на пути до корня в точке a_v. ■

В случае, если в задаче необходим динамический CHT, то есть прямые не упорядочены монотонно по углу наклона, задача становится немного сложнее. Обычно в таком случае прямые хранятся в std::set, и при добавлении по обе стороны от добавляемой прямой удаляются ненужные. Чтобы сделать эту структуру неамортизированной, будем хранить вместо std::set декартово дерево. Тогда спуском по дереву можно найти левую и правую границы отрезка прямых, которые нужно удалить, после чего можно вставить новую прямую. Спуск по дереву в данном случае является заменой стандартному бинпоиску. Чтобы сделать эту структуру персистентной, необходимо использовать персистентное декартово дерево.

В отличие от простого алгоритма для обычного CHT, алгоритм для динамического CHT сложен в реализации, поэтому рекомендуется вместо него использовать все-таки дерево Ли Чао.

14.1 Задачи для практики

- http://ceoi.inf.elte.hu/probarch/09/harbingers.pdf

[2] 9

IV Теория чисел

15 Нахождение обратных ко всем остаткам за $O(p)$ 123

16 Поиск факториала по простому модулю 127

17 Поиск факториала по простому модулю за $O(\sqrt{\min(p,n)}\log^2 n)$ 129

18 Обращение Мёбиуса, свертка Дирихле 131

19 Сумма мультипликативной функции: Powerful Number Sieve 140

20 Квадратный корень по простому модулю за $O(\log p)$ 143

21 Дискретное логарифмирование 145

22 Оценка на количество делителей числа и сверхсоставные числа 146

15. Нахождение обратных ко всем остаткам за $O(p)$

Часто бывает так, что в задаче нужно делить по модулю много раз. Это можно делать обычным алгоритмом взятия обратного по модулю за $O(\log p)$ на запрос. Если мы сделаем n запросов, то асимптотика будет $O(n \log p)$. Сейчас мы рассмотрим алгоритм, который изначально предпосчитает обратные ко всем остаткам за $O(p)$, и тогда на запросы мы будем отвечать за $O(1)$. Если n порядка p или больше, то этот вариант будет более эффективен.

Есть много разных алгоритмов, которые делают это. Здесь будут представлены два, пожалуй, самых простых: один очень простой в понимании и написании, а другой еще легче в написании, однако не настолько очевидный с точки зрения понимания и придумывания.

15.1 Метод обратных факториалов

Идея первого алгоритма заключается в том, что мы посчитаем все возможные факториалы и обратные факториалы, а любое обратное к какому-то остатку представим как отношение двух факториалов.

Теорема 15.1.1 Теорема Вильсона гласит, что если p — простое число, то

$$(p-1)! \equiv -1 \pmod p$$

Доказательство. Давайте заметим, что все остатки от 1 до $p-1$ разбиваются на пары вида x, x^{-1}. Произведение чисел в паре равно единице по модулю p. Есть один крайний случай: когда $x = x^{-1}$. Это происходит в том случае, если $x^2 \equiv 1 \pmod p$, то есть $x^2 - 1 = (x-1) \cdot (x+1) \vdots p$. Значит, $x \equiv \pm 1 \pmod p$. Тогда в итоге $(p-1)!$ по модулю p состоит из произведения нескольких единиц, а также одной -1. Так что $(p-1)! \equiv -1 \pmod p$. Что и требовалось доказать. ■

Зная этот факт, мы можем сразу понять, что $((p-1)!)^{-1} \equiv -1 \pmod p$, потому что обратное к -1 — это -1. Таким образом, обратное к $(p-1)!$ мы уже посчитали.

Замечание 15.1.2 На самом деле не обязательно было пользоваться этой формулой. Можно было посчитать за $O(p)$ число $(p-1)!$, а потом бинарным возведением в степень найти к нему обратное за $O(\log p)$. Итоговая асимптотика бы от этого не пострадала.

Мы уже нашли обратное к $(p-1)!$. Как же найти обратное к $(p-2)!$ теперь? Заметим следующий факт:

$$\frac{1}{k!} = \frac{k+1}{(k+1)!}$$

Так что алгоритм нахождения всех обратных факториалов следующий: идем с конца, изначально устанавливаем, что обратное к $(p-1)!$ — это -1, а затем пересчитываем по очереди обратное к $k!$ как обратное к $(k+1)!$, умноженное на $k+1$.

Теперь пусть мы посчитали все факториалы и все обратные факториалы за $O(p)$. Как найти обратные ко всем остаткам? С этим нам поможет следующая формула:

$$\frac{1}{k} = \frac{(k-1)!}{k!}$$

А отношение двух факториалов — это произведение первого факториала на обратное ко второму.

Представим код алгоритма:

```
1  vector<int> get_all_modular_inverses(int p) {
2      vector<int> inverse_factorials(p);
3      inverse_factorials[p - 1] = p - 1; // -1 mod p = p - 1
4      for (int k = p - 2; k > 0; k--) {
5          inverse_factorials[k] = 1LL * inverse_factorials[k +
               1] * (k + 1) % p;
6      }
7  
8      vector<int> inverses(p);
9      int factorial = 1;
10     for (int k = 1; k < p; k++) {
11         inverses[k] = 1LL * factorial * inverse_factorials[k]
               % p;
12         factorial = 1LL * factorial * k % p;
13     }
14     return inverses;
15 }
```

Весь алгоритм — это два прохода по числам от 1 до $p-1$, так что работает он за $O(p)$. Потребление памяти тоже $O(p)$, потому что нужно хранить массивы факториалов и обратных факториалов. От одного из них можно избавиться, если вычислять ответ на лету (в приведенном коде мы не хранили факториалы), однако не от обоих.

Упражнение 15.1 Придумайте модернизацию этого алгоритма, которая работает за $O(p)$, но при этом потребляет $O(\sqrt{p})$ памяти (считайте, что ответы — вектор `inverses` — вы можете просто выводить на экран, и вам не нужно их хранить). ■

Замечание 15.1.3 Заметим, что можно считать обратные факториалы, начиная не обязательно с $p-1$. Если нам нужно найти обратные ко всем остаткам от 1 до n, то можно за $O(n)$ посчитать $n! \bmod p$, найти к нему обратное за $O(\log p)$ и потом аналогично представленному выше способу насчитать все обратные факториалы от 1 до n. Тогда подсчет обратных ко всем остаткам от 1 до n будет работать за $O(n+\log p)$.

Как вы можете видеть, алгоритм очень простой. Однако его редко получится где-то применить, потому что, во-первых, нахождение всех обратных по отдельности работает за $O(p\log p)$, что тяжело отсечь от $O(p)$ на неучебной задаче, а во-вторых, модуль чаще всего — это число порядка 10^9, поэтому вы не имеете возможности посчитать обратные ко всем остаткам, и использование стандартного алгоритма за $O(n\log p)$ дает более эффективное решение.

15.2 Алгоритм одного цикла

Второй алгоритм пишется всего одним циклом. Однако чтобы его вспомнить, придется написать пару формул на бумажке.

Алгоритм основывается на одном простом факте:

Теорема 15.2.1

$$\frac{1}{k} \equiv -\left\lfloor \frac{p}{k} \right\rfloor \cdot \frac{1}{p \bmod k} \pmod p$$

Доказательство. Давайте представим p в виде $k \cdot x + y$, где $x = \left\lfloor \frac{p}{k} \right\rfloor$ и $y = p \bmod k$.

Необходимо проверить, что

$$k \cdot \left(-\left\lfloor \frac{p}{k} \right\rfloor \cdot \frac{1}{p \bmod k}\right) \equiv 1 \pmod p$$

$$k \cdot (-x \cdot \frac{1}{y}) = -(k \cdot x) \cdot \frac{1}{y} = -((k \cdot x + y) - y) \cdot \frac{1}{y} = -(p-y) \cdot \frac{1}{y} \equiv y \cdot \frac{1}{y} \equiv 1 \pmod p$$

Что и требовалось доказать. ■

Таким образом, мы можем посчитать обратное к k, если уже посчитано обратное к $p \bmod k$. Заметим, что это число меньше, чем k, поэтому все обратные можно вычислять по порядку.

Реализация у этого алгоритма крайне проста:

```
1 vector<int> get_all_modular_inverses(int p) {
2     vector<int> inverses(p);
3     inverses[1] = 1;
4     for (int k = 2; k < p; k++) {
5         inverses[k] = -1LL * (p / k) * inverses[p % k] % p +
             p;
6         // +p because this number is negative
7     }
8     return inverses;
9 }
```

Также преимуществом этого метода является то, что это просто один цикл `for` по возрастанию, поэтому можно считать обратные не ко всем остаткам, а к первым n остаткам за $O(n)$ очень легко. Однако не очень ясно, для чего это может вам понадобиться.

При тестировании на p порядка 10^8 второй алгоритм работает примерно в два раза быстрее, чем первый.

16. Поиск факториала по простому модулю

Часто в задачах бывает необходимо искать $n! \bmod p$. Обычно модуль — это большое число, а n не очень большое, поэтому можно просто заранее предпосчитать все факториалы. Однако иногда бывает обратная ситуация: $p < n$. С первого взгляда кажется, что это бессмысленная задача: в $n!$ в таком случае входит число p, поэтому $n! \bmod p = 0$. Но часто нас не устраивает такой ответ, потому что нам часто нужно делить факториалы друг на друга. В этом случае мы можем получить выражения типа $\frac{0}{0}$, которое на самом деле равно чему-то ненулевому. Встает задача: посчитать $n! \bmod p$ без вхождений p в $n!$ и отдельно посчитать степень вхождения p в факториал, то есть представить $n!$ в виде $a \cdot p^k$, где a не делится на p.

Пускай $\left\lfloor \frac{n}{p} \right\rfloor = k$ и $n \bmod p = b$. Тогда

$$n! = (1 \cdot 2 \cdot \ldots \cdot (p-1)) \cdot p \cdot ((p+1) \cdot (p+2) \cdot \ldots \cdot (2p-1)) \cdot (2p) \cdot \ldots \cdot (kp) \cdot ((kp+1) \cdot (kp+2) \cdot \ldots \cdot (kp+b))$$

Если брать это по модулю p, то получится

$$(p-1)! \cdot p \cdot (p-1)! \cdot (2p) \cdot \ldots \cdot (p-1)! \cdot (kp) \cdot b! = ((p-1)!)^k \cdot b! \cdot (k! \cdot p^k)$$

При этом p^k мы игнорируем, потому что это вхождения p.

По теореме Вильсона (15.1.1) $(p-1)! \bmod p = -1$. Так что нам необходимо посчитать $(-1)^k \cdot b! \cdot k!$. При этом $b < p$ (это остаток от деления на p), так что $b!$ можно посчитать за $O(p)$, после чего рекурсивно запуститься для подсчета $k!$. Каждый раз мы делим число n на p, так что будет всего $\log_p n = \frac{\log n}{\log p}$ итераций. Асимптотика — $O(p \log_p n)$. С другой стороны, все факториалы до p можно предпосчитать заранее, и тогда асимптотика будет $O(\log_p n)$ на запрос и $O(p)$ на предпосчет.

Реализация представлена ниже:

```
1  int mod_factorial(long long n, int p) {
2      int factorial = 1;
3      while (n > 0) {
```

```
4         long long k = n / p;
5         int b = n % p;
6         if (k % 2 == 1) {
7             factorial = 1LL * factorial * (p - 1) % p;
8         }
9         for (int i = 1; i <= b; i++) {
10            factorial = 1LL * factorial * i % p;
11        }
12        n = k;
13    }
14    return factorial;
15 }
```

С другой стороны, если нам надо посчитать степень вхождения p в факториал, это делается еще проще. Давайте для этого посчитаем количество чисел, которые делятся на p, на p^2, на p^3 и т.д. И тогда если число делится ровно на p^k, то мы учтем его как раз k раз. А если заметить, что количество чисел, делящихся на p^i, — это $\left\lfloor \frac{n}{p^i} \right\rfloor$, то получается такой незатейливый алгоритм:

```
1 int factorial_power(int n, int p) {
2     int power = 0;
3     while (n > 0) {
4         n /= p;
5         power += n;
6     }
7     return power;
8 }
```

Асимптотика равна $O(\log_p n)$.

17. Поиск факториала по простому модулю за $O(\sqrt{\min(p,n)}\log^2 n)$

В прошлом разделе мы научились искать $n! \mod p$ без вхождений p в факториал за $O(p\log_p n)$ или за $O(p+\log_p n)$, если заранее предпосчитать все факториалы. Однако чаще всего модуль — это число порядка 10^9, поэтому эти алгоритмы нам не подходят. Давайте научимся искать факториал быстрее.

Давайте сначала научимся искать $n! \mod p$ за $O(\sqrt{n}\log^2 n)$, если $n < p$.

Обозначим $k = \lfloor\sqrt{n}\rfloor$. Тогда мы хотим построить алгоритм за $O(k\log^2 k)$. Пусть $n = k^2 + b$, где $b \leqslant 2\cdot k$. В таком случае нам достаточно посчитать $\left(k^2\right)!$, а затем за $O(k)$ домножить его на b недостающих чисел.

Построим многочлен $P(x) := (kx)\cdot(kx-1)\cdot\ldots\cdot(kx-k+1)$. Тогда $\left(k^2\right)! = P(1)\cdot P(2)\cdot\ldots\cdot P(k)$. Введем еще один многочлен $Q(x) := P(2x)\cdot P(2x-1)$. Получается, что $\left(k^2\right)! = Q(1)\cdot Q(2)\cdot\ldots\cdot Q\left(\frac{k}{2}\right)$, если k четно, а если k нечетно, то еще нужно домножить на $P(k)$, но его мы легко можем сделать за $O(k)$.

Теперь давайте рекурсивно запустимся от многочлена Q. Глубина рекурсии будет $\log k$. Однако проблема в том, что изначально многочлен P имел степень k, однако многочлен Q имеет степень уже $2k$, и степень будет возрастать. Но если учесть, что многочлен Q нам нужно посчитать только в точках $1,2,\ldots,\left\lfloor\frac{k}{2}\right\rfloor$, то можно взять вместо Q многочлен $Q \bmod (x-1)\cdot(x-2)\cdot\ldots\cdot(x-\left\lfloor\frac{k}{2}\right\rfloor)$. И у такого многочлена уже степень будет $< \left\lfloor\frac{k}{2}\right\rfloor$.

Взятие двух многочленов по модулю можно реализовать за $O(k\log^2 k)$ через деление. Тогда асимптотика алгоритма получается $O\left(k\log^2 k + \frac{k}{2}\log^2\left(\frac{k}{2}\right) + \frac{k}{4}\log^2\left(\frac{k}{4}\right) + \ldots\right) = O\left(\left(k + \frac{k}{2} + \frac{k}{4} + \ldots\right)\log^2 k\right) = O(k\log^2 k)$. Что и требовалось.

Чтобы получить алгоритм, который работает при $n \geqslant p$, нужно просто применить алгоритм из предыдущего раздела, однако вычислять $b!$ с помощью нового метода. $b < p$, поэтому это вычисление будет работать не дольше $\sqrt{p}\log^2 p$. Всего итераций будет $\log_p n$, так что асимптотика получается $O(\sqrt{p}\log^2 p\cdot\log_p n) = O\left(\sqrt{p}\log^2 p\frac{\log n}{\log p}\right) = O\left(\sqrt{p}\log p\log n\right)$. Если объединить случаи $p < n$ и $p \geqslant n$, получается время работы $O\left(\sqrt{\min(p,n)}\log\min(p,n)\log n\right)$.

17.1 Задачи для практики

- https://www.spoj.com/problems/FACTMODP/

18. Обращение Мёбиуса, свертка Дирихле

Рассмотрим различные свертки функций. Функции мы будем рассматривать арифметические, то есть действующие из $\mathbb{N}$ в $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$ или $\mathbb{C}$. В каком-то смысле можно думать про арифметические функции как про последовательности чисел: $f(1), f(2), \ldots, f(n), \ldots$.

Возможно, вам уже знакомы некоторые свертки. К примеру, свертка умножения:

$$c_n = \sum_{k=0}^{n} a_k \cdot b_{n-k}$$ / здесь последовательности нумеруются с нуля/

Быстрое преобразование Фурье как раз строит по последовательностям a и b их свертку умножения.

В этом разделе мы рассмотрим несколько примеров других сверток.

18.1 Формула обращения Мёбиуса

Часто бывает так, что две функции связаны между собой следующим отношением:

$$g(n) = \sum_{d|n} f(d)$$

/ Запись $d|n$ означает "d делит n", то есть сумма берется по всем делителям числа n /

Определение 18.1.1 Функция Мёбиуса.

$$\mu(n) = \begin{cases} 0, & \text{если } n \text{ не свободно от квадратов,} \\ (-1)^k, & \text{если } n \text{ свободно от квадратов, и у него ровно } k \text{ простых делителей} \end{cases}$$

/ Число свободно от квадратов, если все простые входят в его разложение в степени ровно один /

■ **Пример 18.1** 1. Количество делителей числа: $\tau(n) = \sum\limits_{d|n} 1$.

2. Сумма делителей числа: $\sigma(n) = \sum\limits_{d|n} d$.

3. Функция, которая равна единице в единице и нулю во всех остальных натуральных числах: $\chi_1 = \sum\limits_{d|n} \mu(d)$. ■

Доказательство. 3. Пусть $n = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdot \ldots \cdot p_k^{\alpha_k}$. Если $\mu(d) \neq 0$, то каждую p-шку мы взяли в него либо в нулевой степени, либо в первой. Тогда есть $\sum\limits_{i=0}^{\lfloor \frac{k}{2} \rfloor} \binom{k}{2i}$ d-шек, для которых $\mu(d) = 1$ и $\sum\limits_{i=0}^{\lfloor \frac{k-1}{2} \rfloor} \binom{k}{2i+1}$ d-шек, для которых $\mu(d) = -1$. А такие суммы равны при $n > 1$ (сумма четных биномиальных коэффициентов равна сумме нечетных), так что вся сумма равна нулю. При $n = 1$ легко произвести подстановку и проверить, что $\chi_1(1) = 1$. ■

Определение 18.1.2 Функция Эйлера $\varphi(n)$ — это количество чисел, не больших n, которые взаимно просты с n.

Замечание 18.1.1 Есть формула для функции Эйлера, которая имеет похожий вид: $\varphi(n) = n \prod\limits_{p|n} (1 - \frac{1}{p})$, где произведение берется по всем простым делителям n.

Теорема 18.1.2 Формула обращения Мёбиуса
Пусть f и g — арифметические функции. Тогда
$g(n) = \sum\limits_{d|n} f(d) \Leftrightarrow f(n) = \sum\limits_{d|n} \mu(d) g\left(\frac{n}{d}\right)$.

Доказательство.

$$\sum_{d|n} \mu(d) g\left(\frac{n}{d}\right) = \text{/ по левой формуле, которая у нас уже есть /} =$$

$$= \sum_{d|n} \mu(d) \left(\sum_{d'|\frac{n}{d}} f(d') \right) = \sum_{d,d':(d \cdot d')|n} \mu(d) \cdot f(d') = \text{/ поменяем местами обозначения /} =$$

$$= \sum_{d,d':(d \cdot d')|n} \mu(d') \cdot f(d) = \sum_{d|n} f(d) \left(\sum_{d'|\frac{n}{d}} \mu(d') \right) = \text{/ по примеру 3 /} = \sum_{d|n} f(d) \chi_1 \left(\frac{n}{d}\right) = f(n)$$

Последнее равенство верно, потому что у всех слагаемых кроме $d = n$ будет множитель 0.

В обратную сторону аналогично. Нужно тоже подставить. ■

■ **Пример 18.2** Подставим в эту формулу примеры выше:

1. $1 = \sum\limits_{d|n} \mu(d) \tau\left(\frac{n}{d}\right)$.
2. $n = \sum\limits_{d|n} \mu(d) \sigma\left(\frac{n}{d}\right)$.

3. $\mu(n) = \sum\limits_{d|n} \mu(d)\chi_1\left(\frac{n}{d}\right)$ (что очевидно, но показывает, что формула из третьего примера является частным случаем формулы обращения Мёбиуса). ■

■ **Пример 18.3** Существует такое тождество: $n = \sum\limits_{d|n} \varphi(d)$. Применим формулу обращения Мёбиуса и получим, что $\varphi(n) = \sum\limits_{d|n} \mu(d) \cdot \frac{n}{d}$.

Неожиданно получили связь функций Эйлера и Мёбиуса. ■

18.2 Свертка Дирихле

$\sum\limits_{d|n} \mu(d)g\left(\frac{n}{d}\right)$ является частным случаем свертки Дирихле:

Определение 18.2.1 Свертка Дирихле двух арифметических функций определяется следующим образом:
$(f*g)(n) = \sum\limits_{d|n} f(d)g\left(\frac{n}{d}\right) = \sum\limits_{ab=n} f(a)g(b)$.

Замечание 18.2.1 Тогда формулу обращения Мёбиуса можно коротко записать как

$$g = f*1 \Leftrightarrow f = g*\mu$$

Определение 18.2.2 Арифметическая функция f называется мультипликативной, если $f(a \cdot b) = f(a) \cdot f(b)\ \forall\ a,b \in \mathbb{N}$, таких что $gcd(a,b) = 1$.

Замечание 18.2.2 $\mu, \varphi, \tau, \sigma, \chi_1, id, 1$ — мультипликативные арифметические функции. / 1 — это фунция, равная единице во всех точках /

Свойства 18.1 1. $(f*g)*h = f*(g*h)$.
2. $f*g = g*f$.
3. $f*(g+h) = f*g+f*h$.
4. $f*\chi_1 = \chi_1*f = f$.
5. $f*0 = 0*f = 0$ (0 — это функция, которая всегда равна нулю).
6. Если f и g — мультипликативные, то $f*g$ тоже является мультипликативной.
■

Доказательство. 1-5. Очевидно. Проверяется подстановкой.

6. $gcd(n,m) = 1$, $(f*g)(n \cdot m) = \sum\limits_{d|n \cdot m} f(d)g(\frac{n \cdot m}{d}) =$ / здесь мы пользуемся тем, что числа взаимно просты; иначе мы бы получили сумму с повторениями / $= \sum\limits_{d_1|n, d_2|m} f(d_1 \cdot d_2)g(\frac{n \cdot m}{d_1 \cdot d_2}) =$ / а здесь тем, что числа делители взаимно простых чисел взаимно просты /

$$= \sum_{d_1|n, d_2|m} f(d_1)f(d_2)g(\tfrac{n}{d_1})g(\tfrac{m}{d_2}) = \left(\sum_{d_1|n} f(d_1)g(\tfrac{n}{d_1})\right) \cdot \left(\sum_{d_2|m} f(d_2)g(\tfrac{m}{d_2})\right) = (f*g)(n) \cdot (f*g)(m).$$

■

Получили, что арифметические функции образуют коммутативное кольцо с единицей относительно свертки Дирихле и поточечного сложения. Нулем в этом кольце является тождественно нулевая функция, а единицей — функция χ_1. Это кольцо называется **кольцом Дирихле**.

■ **Пример 18.4** a. $\tau = 1 * 1$.
b. $\sigma = id * 1$.
c. $\chi_1 = 1 * \mu$.
d. $1 = \tau * \mu$.
e. $id = \sigma * \mu$.
f. $id = \varphi * 1$.
g. $\sigma = \varphi * \tau$.

■

Определение 18.2.3 Обращением Дирихле функции f назовем такую функцию g, что $f * g = \chi_1$, то есть функция $g = f^{-1}$ в кольце Дирихле.

Теорема 18.2.3 $\forall f : f(1) \neq 0 \ \exists g : f * g = \chi_1$.

Доказательство. $g(1) = \frac{1}{f(1)}$.
А для $n > 1$ функцию g можно вычислить рекурсивно:
$g(n) = -\frac{1}{f(1)} \sum\limits_{d|n, d \neq n} f(\frac{n}{d}) g(d)$.

Доказывается, что такая функция подходит, домножением на $f(1)$ и переносом всего в левую часть. ■

18.3 Поиск количества пар взаимнопростых, не больших n

Зачем это все нужно? Рассмотрим, как решать задачи при помощи обращения Мёбиуса и свертки Дирихле.

Обозначение 18.3.1 $[P]$ — это функция, которая равна единице, если P верно, и нулю, если P неверно.

Давайте сначала разберемся, как пользоваться линейным решетом Эратосфена. Можно считать мультипликативную функцию f для всех чисел от 1 до n за $O(n)$, если за $O(1)$ можно находить $f(p^k)$.

■ **Пример 18.5** Дано число n. Надо найти количество упорядоченных пар взаимно простых чисел $x, y \leqslant n$. ■

Обозначим ответ за $f(n)$.

Сначала заметим, что $f(n) = 2 \cdot \sum\limits_{k=1}^{n} \varphi(k) - 1$, потому что каждую неупорядоченную пару x, y мы посчитаем один раз в $\varphi(max(x,y))$, а нам надо посчитать упорядоченные пары, поэтому необходимо умножить на 2. Осталось вычесть пары из двух одинаковых чисел. Но число взаимно просто с собой только если оно равно единице. Этой формулы нам уже хватит на самом деле, но давайте придумаем альтернативную, потому что в более сложных случаях все будет аналогично.

Запишем формулу: $f(n) = \sum\limits_{i=1}^{n} \sum\limits_{j=1}^{n} [gcd(i,j) = 1]$. Заметим, что $[gcd(i,j) = 1] = \chi_1(gcd(i,j))$, так что можно применить свертку Мёбиуса:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{d|gcd(i,j)} \mu(d) = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{d=1}^{n} [d|gcd(i,j)] \cdot \mu(d) =$$

$$= \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{d=1}^{n} [d|i] \cdot [d|j] \cdot \mu(d) = \text{/ меняем порядок суммирования /} =$$

$$= \sum_{d=1}^{n} \mu(d) \left(\sum_{i=1}^{n} [d|i] \right) \left(\sum_{j=1}^{n} [d|j] \right)$$

При этом $\sum\limits_{i=1}^{n} [d|i] = \sum\limits_{j=1}^{n} [d|j] =$ количеству чисел, делящихся на d, то есть $\lfloor \frac{n}{d} \rfloor$. Тогда получается, что $f(n) = \sum\limits_{d=1}^{n} \mu(d) \cdot \lfloor \frac{n}{d} \rfloor^2$.

А эту формулу можно посчитать уже за линейное время (все значения μ считаются при помощи линейного решета за $O(n)$).

Нырнем глубже! Мы хотим еще быстрее. Пусть у нас есть некоторая мультипликативная функция f, и мы хотим посчитать ее префиксную сумму. Обозначим $s_f(n) = \sum\limits_{k=1}^{n} f(k)$. Предположим, что мы обнаружили такую мультипликативную функцию g, что s_g и s_{f*g} можно быстро считать. Тогда научимся быстро считать s_f:

$$s_{f*g}(n) = \sum_{k=1}^{n} \sum_{d|k} g(d) f\left(\frac{k}{d}\right) = \sum_{d=1}^{n} g(d) \sum_{k=1}^{\lfloor \frac{n}{d} \rfloor} f(k) =$$

$$= g(1) \sum_{k=1}^{n} f(k) + \sum_{d=2}^{n} g(d) \sum_{k=1}^{\lfloor \frac{n}{d} \rfloor} f(k) = g(1) s_f(n) + \sum_{d=2}^{n} g(d) s_f\left(\left\lfloor \frac{n}{d} \right\rfloor\right)$$

Теперь перенесем $s_f(n)$ в одну часть, а все остальное в другую:

$$s_f(n) = \frac{s_{f*g}(n) - \sum_{d=2}^{n} s_f\left(\lfloor \frac{n}{d} \rfloor\right) g(d)}{g(1)}$$

Замечание 18.3.1 Обратите внимание, что если g — мультипликативная функция, то $g(k) = g(1) \cdot g(k)$ для любого k, поэтому если g — это не тождественный ноль, то $g(1) = 1$, так что делить на знаменатель нам не придется.

Осталось научиться быстро считать сумму в числителе.

Лемма 18.3.2 Среди чисел $\lfloor \frac{n}{1} \rfloor, \lfloor \frac{n}{2} \rfloor, \ldots, \lfloor \frac{n}{n} \rfloor$ не более $2\sqrt{n}$ различных.

Доказательство. Стандартное доказательство. Есть $\sqrt{n}$ чисел вида $\lfloor \frac{n}{d} \rfloor$, где $d < \sqrt{n}$, а для $d \geqslant \sqrt{n}$ будет выполнено $\lfloor \frac{n}{d} \rfloor \leqslant \sqrt{n}$, поэтому среди них тоже не больше, чем $\sqrt{n}$ различных.

Также есть простой алгоритм перебора отрезков одинаковых значений $\lfloor \frac{n}{d} \rfloor$:

```
1 for (int left = 1, right; left <= n; left = right) {
2     right = n / (int)(n / left) + 1;
3     // in [left; right) values of n/d are equal
4 }
```

■

Из леммы следует, что сумма в числителе разбивается на $O(\sqrt{n})$ рекурсивных вызовов $s_f(\lfloor \frac{n}{d} \rfloor)$ и подсчетов g на отрезке, но сумма g на отрезке — это разность двух префиксных сумм, которые мы по предположению задачи умеем быстро считать.

Получился рекурсивный алгоритм. Соптимизируем его, сохраняя уже посчитанные значения $s_f(k)$ в хеш-таблицу, чтобы не считать их заново.

Лемма 18.3.3 Пусть a, b и c — натуральные числа. Тогда

$$\left\lfloor \frac{\left\lfloor \frac{a}{b} \right\rfloor}{c} \right\rfloor = \left\lfloor \frac{a}{bc} \right\rfloor$$

Доказательство. Очевидно, что $0 \leqslant \frac{a}{b} - \left\lfloor \frac{a}{b} \right\rfloor < 1$, поэтому $0 \leqslant \frac{a}{bc} - \frac{\left\lfloor \frac{a}{b} \right\rfloor}{c} < \frac{1}{c}$. При этом $\left\lfloor \frac{a}{b} \right\rfloor$ — целое число, поэтому дробная часть $\frac{\left\lfloor \frac{a}{b} \right\rfloor}{c}$ не больше $\frac{c-1}{c}$, так что при прибавлении чего-то меньшего, чем $\frac{1}{c}$, мы не перепрыгнем на следующую целую часть. ■

Из леммы слеудет, что за время алгоритма мы посетим только числа вида $\left\lfloor \frac{n}{d} \right\rfloor$, потому что $\left\lfloor \frac{\left\lfloor \frac{n}{d_1} \right\rfloor}{d_2} \right\rfloor = \left\lfloor \frac{n}{d_1 d_2} \right\rfloor$.

Подсчет $s_f(k)$ занимает $O(\sqrt{k})$ времени + рекурсивные вызовы. Так что итоговая асимптотика будет равна

$$O\left(\sum_{k=1}^{n} \sqrt{\left\lfloor \frac{n}{k} \right\rfloor}\right) \text{ / сумма берется по различным значениям / } \leqslant O\left(\sum_{k=1}^{\sqrt{n}} \sqrt{k} + \sum_{k=1}^{\sqrt{n}} \sqrt{\frac{n}{k}}\right)$$

Когда в асимптотике есть сумма возрастающей/убывающей функции, эту сумму можно заменить на интеграл без потерь.

$\sum\limits_{k=1}^{\sqrt{n}} \sqrt{k} \sim \int\limits_{1}^{\sqrt{n}} \sqrt{k}dk = \frac{2}{3}k^{\frac{3}{2}}|_1^{\sqrt{n}} = O(n^{\frac{3}{4}})$

$\sum\limits_{k=1}^{\sqrt{n}} \sqrt{\frac{n}{k}} \sim \int\limits_{1}^{\sqrt{n}} \sqrt{\frac{n}{k}}dk = 2\sqrt{nk}|_1^{\sqrt{n}} = O(n^{\frac{3}{4}})$

Получили, что асимптотика — $O(n^{\frac{3}{4}})$.

Самое время вспомнить о том, что функция f мультипликативная. А для мультипликативных функций мы умеем считать ее первые k значений за $O(k)$ при помощи линейного решета Эратосфена. Тогда и префиксные суммы мы тоже можем посчитать за $O(k)$. Пусть мы предпосчитали префиксные суммы для первых $k \geqslant \sqrt{n}$ чисел. Тогда нам надо брать сумму времени работы только для таких $\left\lfloor \frac{n}{d} \right\rfloor$, которые больше k. Получаем асимптотику $O(k + \sum\limits_{i=1}^{\frac{n}{k}} \sqrt{\frac{n}{i}}) = O(k + \frac{n}{\sqrt{k}})$ (этот интеграл мы уже брали, надо подставить только в другой точке). Минимума эта величина достигает при $k = O(n^{\frac{2}{3}})$ (сумма возрастающей и убывающей функций достигает асимптотического минимума в точке пересечения). В этом случае асимптотика получается равна $O(n^{\frac{2}{3}})$.

Ура, мы научились искать префиксную сумму мультипликативной функции f за $O(n^{\frac{2}{3}})$, если есть такая функция g, что s_g и s_{f*g} мы умеем считать за $O(1)$.

Откуда же взять эту функцию g? Мы знаем несколько примеров таких функций: $\chi_1, id, 1$ и так далее. Надо поперебирать эти функции, пока их свертка с f не получится равной тоже какой-то простой функции. Если не нашлось, то очень жаль, ничего не получилось.

■ **Пример 18.6** Вернемся к нашей задаче поиска количества упорядоченных пар взаимно простых чисел $x, y \leqslant n$.

Мы уже выяснили, что $f(n) = \sum\limits_{d=1}^{n} \mu(d) \left\lfloor \frac{n}{d} \right\rfloor^2$. Как мы уже поняли, среди чисел вида $\left\lfloor \frac{n}{d} \right\rfloor$ всего $\sqrt{n}$ различных. А μ является мультипликативной функцией, при этом $\mu * 1 = \chi_1$. Так что сумму μ на отрезке мы умеем считать быстро. Посмотрим, в каких точках m нам надо будет считать $s_\mu(m)$. Это будут такие m, что на них число $\left\lfloor \frac{n}{m} \right\rfloor$ увеличилось. Нетрудно понять, что это будут как раз числа вида $\left\lfloor \frac{n}{k} \right\rfloor$. Поэтому нам надо будет посчитать s_μ ровно в точках вида $\left\lfloor \frac{n}{k} \right\rfloor$, а это мы умеем делать суммарно за $O(n^{\frac{2}{3}})$. ■

■ **Пример 18.7** $\varphi * 1 = id$. Так что s_φ тоже можно считать за $O(n^{\frac{2}{3}})$. И в действительности в данном случае нам этого достаточно, потому что мы с самого начала выразили количество пар взаимно простых через префиксные суммы φ. ■

18.4 Сумма попарных НОДов чисел, не больших n

■ **Пример 18.8** Дано число n. Надо найти

$$\sum_{i=1}^{n} \sum_{j=1}^{n} gcd(i, j)$$

То есть сумму попарных НОДов всех чисел, которые не больше n. ■

Давайте воспользуемся похожими на пример с парами взаимно простых свойствами. Давайте перебирать этот самый НОД d от 1 до n и считать количество пар чисел, у которых НОД равен d. Если НОД равен d, то точно оба числа i и j делятся на d, однако если мы поделим их на d и получим числа i' и j', то эти два числа уже взаимно просты. Поэтому на самом деле количество пар чисел i и j, НОД которых равен d, совпадает с количеством пар чисел i' и j', которые не больше $\left\lfloor \frac{n}{d} \right\rfloor$, и при этом взаимно просты. А это как раз $f(\left\lfloor \frac{n}{d} \right\rfloor)$ в обозначениях прошлого примера. То есть формула получается такая:

$$\sum_{d=1}^{n} d \cdot f(\left\lfloor \frac{n}{d} \right\rfloor)$$

Как и раньше, мы знаем, что $\left\lfloor \frac{n}{d} \right\rfloor$ меняется не очень часто, поэтому есть $\sqrt{n}$ отрезков равенства, на которых нужно один раз посчитать f, а также высчитать сумму d на отрезке. Так же, как и в примере с μ в прошлый раз, вызов $f(n)$ посчитает сразу же все нужные нам значения f, поэтому итоговая асимптотика будет $O(n^{\frac{2}{3}})$.

18.5 Сумма попарных НОКов чисел, не больших n

■ **Пример 18.9** Дано число n. Надо найти

$$\sum_{i=1}^{n} \sum_{j=1}^{n} lcm(i, j)$$

То есть сумму попарных НОКов всех чисел, которые не больше n. ■

В этом нам поможет следующий общеизвестный факт:

Замечание 18.5.1

$$a \cdot b = gcd(a, b) \cdot lcm(a, b)$$

Давайте запишем нашу сумму как обычно:

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{c=1}^{n} c\cdot[lcm(i,j)=c] = \text{/ По замечанию 18.5.1 /} = \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{d=1}^{n}\frac{i\cdot j}{d}\cdot[gcd(i,j)=d]=$$

$$= \text{/ } i'=\frac{i}{d} \text{ и } j'=\frac{j}{d} \text{ /} = \sum_{d=1}^{n}\sum_{i'=1}^{\lfloor\frac{n}{d}\rfloor}\sum_{j'=1}^{\lfloor\frac{n}{d}\rfloor}\frac{i'\cdot d\cdot j'\cdot d}{d}\cdot[gcd(i',j')=1] = \sum_{d=1}^{n}\sum_{i'=1}^{\lfloor\frac{n}{d}\rfloor}\sum_{j'=1}^{\lfloor\frac{n}{d}\rfloor} i'\cdot d\cdot j'\cdot\chi_1(gcd(i',j'))=$$

$$= \text{/ по формуле для } \chi_1\text{, доказанной в 18.1 /} = \sum_{d=1}^{n}\sum_{i'=1}^{\lfloor\frac{n}{d}\rfloor}\sum_{j'=1}^{\lfloor\frac{n}{d}\rfloor} i'\cdot d\cdot j' \sum_{k|gcd(i',j')}\mu(k)=$$

$$= \sum_{d=1}^{n}\sum_{i'=1}^{\lfloor\frac{n}{d}\rfloor}\sum_{j'=1}^{\lfloor\frac{n}{d}\rfloor} i'\cdot d\cdot j' \sum_{k=1}^{\lfloor\frac{n}{d}\rfloor}[k|i']\cdot[k|j']\cdot\mu(k)=$$

$$= \sum_{d=1}^{n} d\sum_{k=1}^{\lfloor\frac{n}{d}\rfloor}\mu(k)\cdot\left(\sum_{i'=1}^{\lfloor\frac{n}{d}\rfloor} i'\cdot[k|i']\right)\cdot\left(\sum_{j'=1}^{\lfloor\frac{n}{d}\rfloor} j'\cdot[k|j']\right) = \star$$

Оба выражения в скобках равны сумме чисел $\leqslant \lfloor\frac{n}{d}\rfloor$, которые делятся на k. Это числа $k, 2k, \ldots, \left\lfloor\frac{\lfloor\frac{n}{d}\rfloor}{k}\right\rfloor\cdot k = \lfloor\frac{n}{dk}\rfloor\cdot k$. Их сумма равна

$$k\cdot\frac{(\lfloor\frac{n}{dk}\rfloor)\cdot(\lfloor\frac{n}{dk}\rfloor+1)}{2}$$

Обозначим второй множитель за $g(\lfloor\frac{n}{dk}\rfloor)$
Подставим это в нашу формулу:

$$\star = \sum_{d=1}^{n} d\sum_{k=1}^{\lfloor\frac{n}{d}\rfloor}\mu(k)\cdot k^2\cdot g^2\left(\left\lfloor\frac{n}{dk}\right\rfloor\right) = \text{/ обозначим } l=kd \text{ /} =$$

$$= \sum_{l=1}^{n} l\cdot g^2\left(\left\lfloor\frac{n}{l}\right\rfloor\right)\sum_{k|l} k\cdot\mu(k) = \text{/ } h(l):=l\cdot\sum_{k|l}k\cdot\mu(k) \text{ /} = \sum_{l=1}^{n} g^2\left(\left\lfloor\frac{n}{l}\right\rfloor\right)\cdot h(l)$$

При этом h можно выразить как $id\cdot(1*(id\cdot\mu))$, так что h мультипликативна, потому что и свертка, и произведение мультипликативных функций являются мультипликативными. Кроме того, $g(n,l)$ вычисляется по формуле, поэтому ее мы можем считать за $O(1)$. Так что такую сумму можно считать за $O(n)$, так как мультипликативную функцию h можно насчитать линейным решетом ($h(p^k) = p^k\cdot(1-p)$ вычисляется за $O(1)$).

Однако мы хотим быстрее. В g мы подставляем всего $\sqrt{n}$ различных значений, поэтому как раньше, нам надо научиться быстро считать s_h. Для этого нужно найти какую-то мультипликативную функцию q, чтобы s_q и s_{h*q} можно было считать за $O(1)$.

Давайте докажем, что $h*id^2 = id$. В этом нам поможет следующий факт:

Лемма 18.5.2 Для любых арифметических функций f, g верно:

$$(id\cdot f)*(id\cdot g) = id\cdot(f*g)$$

Доказательство. Это доказывается простой подстановкой:

$$((id \cdot f) * (id \cdot g))(n) = \sum_{d|n} d \cdot f(d) \cdot \frac{n}{d} \cdot g\left(\frac{n}{d}\right) = n \cdot \sum_{d|n} f(d) \cdot g\left(\frac{n}{d}\right) = n \cdot (f * g)$$

■

Мы хотим доказать, что $h * id^2 = id$. Запишем цепочку равенств:

$$h * id^2 = (id \cdot (1 * (id \cdot \mu))) * (id \cdot id) = \text{/ по лемме /} = id \cdot ((1 * (id \cdot \mu)) * id) =$$

$$= \text{/ по ассоциативности свертки /} = id \cdot (1 * ((id \cdot \mu) * (id \cdot 1))) = \text{/ по лемме /} = id \cdot (1 * (id \cdot (\mu * 1))) =$$

$$= \text{/ по примеру с. из 18.4 /} = id \cdot (1 * (id \cdot \chi_1)) = \text{/ } \chi_1 \text{ не равна нулю только в единице, а } id(1) = 1 \text{ /} =$$

$$= id \cdot (1 * \chi_1) = \text{/ по свойству 4 из 18.1 /} = id \cdot 1 = id$$

Что и требовалось доказать. При этом s_{id} и s_{id^2} можно легко вычислять за $O(1)$:

$$s_{id}(n) = \frac{n \cdot (n+1)}{2}$$

$$s_{id^2}(n) = \frac{n \cdot (n+1) \cdot (2n+1)}{6}$$

Так что мы научились искать сумму НОКов всех пар чисел, не больших n, за $O(n^{\frac{2}{3}})$.

18.6 Задачи для практики

- https://www.codechef.com/NOV15/problems/SMPLSUM
- http://acm.hdu.edu.cn/showproblem.php?pid=1695
- http://poj.org/problem?id=3904
- http://acm.hdu.edu.cn/showproblem.php?pid=5608
- https://www.spoj.com/problems/DIVCNT2/

19. Сумма мультипликативной функции: Powerful Number Sieve

Автор главы: Александр Голованов

Задача 19.1 Дана мультипликативная функция $f(n)$, а также число N. Необходимо посчитать $s_f(N) = \sum_{k=1}^{N} f(k)$.

Мы умеем делать это за $O(n^{2/3})$, если нам известна мультипликативная функция g, для которой мы умеем считать s_g и s_{f*g} за $O(1)$[1]. Оказывается, зачастую это можно делать быстрее, если функции обладают некоторыми дополнительными свойствами.

Пусть $g(n)$ и $h(n)$ — такие мультипликативные функции, что $f = h * g$. Тогда верна следующая лемма:

Лемма 19.0.1

$$s_f(n) = \sum_{k=1}^{n} h(k) s_g\left(\left\lfloor \frac{n}{k} \right\rfloor\right)$$

Доказательство. В сумме слева есть все возможные слагаемые вида $h(k) \cdot g(l)$, где $k \cdot l \leqslant n$. Таким образом, для фиксированного k мы должны перемножить $h(k)$ со всеми такими $g(l)$, что $l \leqslant \left\lfloor \frac{n}{k} \right\rfloor$, то есть с префиксной суммой g. Что и требовалось доказать. ■

Пускай функция h такова, что $h(p) = 0$ для любого простого p. Тогда заметим, что в правой части выражения из леммы те слагаемые, в которых k содержит в себе некоторый простой множитель ровно в первой степени, зануляются (из свойства функции h). Остальные же числа содержат каждый свой простой множитель хотя бы во второй степени. Будем называть такие числа *сильными* (от английского *powerful*, которое происходит от слова *power* в значении «степень»).

Лемма 19.0.2 Всякое сильное число n можно представить в виде a^2b^3 для некоторых натуральных a и b.

[1] 18

Доказательство. Для этого нужно представить каждую степень простого $p_i^{\alpha_i}$, входящую в n, в таком виде, а потом перемножить их. То есть надо просто представить степень α_i в виде $2x+3y$. Безусловно, это можно сделать, так как $\alpha_i \geqslant 2$ (для четного n можно взять $x = \frac{n}{2}$ и $y = 0$, а для нечетного подойдет решение $x = \frac{n-3}{2}$ и $y = 1$). ■

Тогда для вычисления s_f нам нужно научиться эффективно перебирать все сильные числа k и суммировать $h(k)s_g\left(\left\lfloor\frac{n}{k}\right\rfloor\right)$. Будем делать это рекурсивно, перебирая простые числа по возрастанию и выбирая их в степени либо ноль, либо хотя бы два. Обратите внимание на то, что если простое число p входит в разложение сильного числа k, то $p \leqslant \sqrt{k} \leqslant \sqrt{n}$, потому что каждый простой множитель входит в k хотя бы во второй степени, поэтому перебирать простые мы можем только до $\sqrt{n}$.

Рассмотрим следующий алгоритм вычисления функции s_f:

```
1  T s_f(long long n, long long cur, int idx, bool new_value =
      true) {
2    T ans = new_value ? h(cur) * s_g(n / cur) : 0;
3    if (n / cur / primes[idx] / primes[idx] == 0) { // can't
        multiply anymore
4       return ans;
5    }
6    ans += s_f(n, cur, idx + 1, false);
7    cur = cur * primes[idx] * primes[idx];
8    while (cur <= n) {
9        ans += s_f(n, cur, idx + 1, true);
10       cur *= primes[idx];
11   }
12   return ans;
13 }
```

Здесь `primes` — массив с простыми числами хотя бы до $\sqrt{N}$; `cur` — текущее значение k, которое мы перебираем рекурсивно; `idx` — индекс текущего простого из списка `primes`; `new_value` — флаг, который нужен, чтобы мы не прибавили одно и то же значение к ответу несколько раз, когда мы не берем текущее простое в разложение; а тип `T` может быть любым типом на Ваш выбор: например, `long long` или Ваш класс для модульной арифметики.

Легко видеть, что этот алгоритм вычисляет s_f, перебирая лишь сильные числа. Оценим время его работы. Будем считать, что значение функции s_g в точке n мы можем вычислить за время $T_{s_g}(n)$, а вычисление $h(cur)$ работает за $O(1)$ (оно может считаться на лету по мультипликативности через значения в степенях простых). Тогда

$$T_{s_f(n)} = \sum_{k \text{ is powerful}} T_{s_g}\left(\left\lfloor\frac{N}{k}\right\rfloor\right) \leq \sum_{a,b:a^2b^3 \leq N} T_{s_g}\left(\frac{N}{a^2b^3}\right) \approx \int\limits_1^{\sqrt{N}} \int\limits_1^{\sqrt[3]{N/a^2}} T_{s_g}\left(\frac{N}{a^2b^3}\right) \mathrm{d}b\,\mathrm{d}a$$

Для простоты предположим, что $T_{s_g}(n) = n^c$, где $c > 1/3$. Тогда

$$\int_1^{\sqrt{N}} \int_1^{\sqrt[3]{N/a^2}} T_{s_g}\left(\frac{N}{a^2b^3}\right) \mathrm{d}b\,\mathrm{d}a = \int_1^{\sqrt{N}} \int_1^{\sqrt[3]{N/a^2}} \frac{N^c}{a^{2c}b^{3c}} \mathrm{d}b\,\mathrm{d}a = N^c \int_1^{\sqrt{N}} a^{-2c} \int_1^{\sqrt[3]{N/a^2}} b^{-3c}\,\mathrm{d}b\,\mathrm{d}a =$$

$$= N^c \int_1^{\sqrt{N}} a^{-2c} \cdot 1/(3c-1) \cdot (1 - const_1)\,\mathrm{d}a \leqslant$$

$$\leqslant 1/(3c-1)N^c \int_1^{\sqrt{N}} a^{-2c}\,\mathrm{d}a \leq \begin{cases} 1/((1-2c)(3c-1))N^c \cdot N^{1/2-c} = O(\sqrt{N}), & c < \frac{1}{2}, \\ 1/(3c-1)N^c \ln\sqrt{N} = O(\sqrt{N}\log n), & c = \frac{1}{2}, \\ 1/((3c-1)(2c-1))N^c = O(T_{s_g}(N)), & c > \frac{1}{2}. \end{cases}$$

Где $const_1$ — какое-то положительное число.

Очевидно, если $c \leqslant 1/3$, то время работы и подавно составляет $O(\sqrt{N})$.

■ **Пример 19.1** Если $f(n) = \mathrm{rad}(n)$, то есть $f(p_1^{\alpha_1} \cdot \ldots \cdot p_k^{\alpha_k}) = p_1 p_2 \cdot \ldots \cdot p_k$, то в качестве $h(n)$ можно взять мультипликативную функцию, для которой верно $h(1) = 1$, $h(p) = 0$ и $h(p^k) = p - p^2$ для всех $k > 1$, а в качестве g можно взять $g(n) = n$. Легко проверить, что $f = g * h$. Тогда сумма $f(n)$ на префиксе размера N будет считаться за $O(\sqrt{N})$, потому что $T_{s_g}(n) = O(1)$. ■

Замечание 19.0.3 Как видно из примера, при поиске функций h и g имеет смысл подобрать их значения на степенях простых.

Замечание 19.0.4 При оценке времени работы мы пользовались представимостью сильных чисел в виде a^2b^3. Казалось бы, можно было бы написать два for-а по числам a и b, и такой алгоритм был бы проще рекурсивного перебора сильных чисел. Однако сильные числа бывают представимы в таком виде больше, чем одним способом, поэтому при таком алгоритме придётся каким-то образом проверять числа на уникальность, что может повлиять на время работы.

20. Квадратный корень по простому модулю за $O(\log p)$

Задача формулируется следующим образом. Даны числа a и p. При этом p — простое. Нужно найти такое z, что $z^2 \mod p = a$ или сказать, что такого z не существует.

Давайте опишем алгоритм.

Если $p = 2$ или $a = 0$, то $z = a$.

Иначе $p \geqslant 3$, $a \geqslant 1$.

Если $a^{(p-1)/2} \neq 1 \mod p$, то ответа не существует. Иначе же ответ есть.

Запускаем бесконечный цикл, пока не найдем ответ. В нем выбираем i случайным образом из чисел $1, 2, \ldots, p-1$. Считаем многочлен $T(x) := (x+i)^{(p-1)/2} - 1 \mod (x^2 - a) = bx + c$ бинарным возведением в степень и взятием по модулю каждый раз.

После чего если $b \neq 0$, то возьмем $z' = c/b = c \cdot b^{p-2} \mod p$ и проверим, подходит ли оно (возведем в квадрат). Если подходит, вернем, иначе продолжаем перебирать i.

Почему этот алгоритм работает? Крайние случаи очевидны. Иначе, числа, у которых есть корень, называются квадратичными вычетами. Давайте заметим, что у каждого числа ровно 2 корня, либо ровно 0 (если есть корень, то подходит и минус корень, а больше быть не может, потому что $z^2 = t^2 \mod p$ решается только так). То есть у нас есть ровно $(p-1)/2$ чисел, являющихся квадратичными вычетами и $(p-1)/2$, не являющихся. $x^{p-1} = 1 \mod p$, так что для проверки, является ли a квадратом, можно возвести в $(p-1)/2$ степень. То есть проверка на несуществование ответа верная. Теперь осталось его найти!

Пусть $z^2 = a \mod p$. Корни многочлена $x^2 - a$ — это как раз z и $-z$.

В каком случае они будут являться корнями многочлена T? Если $z + i$ и $-z + i$ являются квадратичными вычетами. Нас интересует ситуация, когда ровно один из них — квадратичный вычет. Пусть это $z + i$. Тогда z — корень обоих многочленов, так что после взятия по модулю ответ делится на $x - z$. При этом ответ — многочлен не более, чем первой степени. Так что он имеет вид $dx - dz$. Тогда если мы поделим коэффициенты, то получим как раз z. Аналогично, если $-z$ — корень обоих многочленов. Обратите внимание, что $d \neq 0$, потому что в этом случае оба числа $\pm z$ являются корнями.

Осталось понять, почему этот алгоритм будет работать быстро. Итерация работает за $O(\log p)$. Докажем, что вероятность успеха — $1/2$, тогда матожидание количества шагов будет равно двум.

Если ровно одно из чисел $z+i$ и $-z+i$ является квадратичным вычетом, то $(z+i)^{(p-1)/2} \neq (-z+i)^{(p-1)/2} \mod p$. При этом одно из этих чисел — 1, а другое — -1. Поделим одну часть на другую. $((z+i)/(-z+i))^{(p-1)/2} = -1 \mod p$. То есть это верно, если отношение — не вычет. Докажем, что все такие числа вида $(z+i)/(-z+i)$ различны для всех i, тогда это будет как раз биекция во все вычеты, так что вероятность равна $1/2$.

Пусть $(z+i)/(-z+i) = (z+j)/(-z+j) \mod p$. Домножим на знаменатели, сократим, получим, что $2iz = 2jz \mod p$, то есть $i = j \mod p$.

Вероятность не совсем $1/2$, потому что нужно, чтобы $z+i \neq 0$ и $-z+i \neq 0$. Так что нам не подходят еще два вычета. Алгоритм от этого не страдает, просто для малых p вероятность становится чуть меньше. Можно доказать, что для $p = 3$ все равно все работает.

21. Дискретное логарифмирование

Даны числа a, b и m. Необходимо найти такой x, что $a^x = b \mod m$. При этом $gcd(a,m) = 1$, но m не обязано быть простым.

Ответ есть не всегда, к примеру, при $a = 1$ и $b \neq 1$.

Сделаем корневую декомпозицию. Известно, что $x \in [0, m-1]$. Пусть $k = \lfloor \sqrt{m} \rfloor$. Посчитаем числа вида a^{kn} для $0 \leqslant n \leqslant \lceil \sqrt{m} \rceil$ и положим их в `unordered_map`. Пусть ответ — это x. $x = ki - j$, где $0 \leqslant j < k$.

Тогда равенство можно записать как $a^{ki} = b \cdot a^j \mod p$. Числа слева у нас сохранены. Осталось последовательно домножать b на a, пока такое число не попадется среди сохраненных.

Получается асимптотика $O(\sqrt{m})$.

Это хорошо, но на выполнение n запросов уйдет $O(n\sqrt{m})$ времени.

Давайте делать лучше. Пусть m — это 2, 4, p^k или $2 \cdot p^k$. Найдем заранее g — первообразный корень по модулю m (он как раз существует ровно для таких m) и число $\varphi(m)$. Это можно сделать за $\sqrt{m}$. Затем вместо нахождения логарифма b с основанием a, мы будем искать логарифмы обоих чисел по основанию g, а потом их надо будет просто поделить друг на друга по модулю $\varphi(m)$.

Заметим, что если мы всегда логарифмируем с фиксированным основанием, то первый шаг, на котором мы насчитываем все числа вида g^{kn} можно не выполнять каждый раз, а только один раз в самом начале. Тогда можно взять $k \neq \sqrt{m}$.

Первая фаза работает за m/k и выполняется один раз. Тогда фаза подсчета работает за $O(k)$. Асимптотика — $O(m/k + nk)$. Возьмем $m/k = nk$ для оптимальной асимптотики. $k = \sqrt{m/n}$. Время работы $O(\sqrt{mn} + n \log m)$.

22. Оценка на количество делителей числа и сверхсоставные числа

22.1 Предыстория и мотивация

Иногда бывает так, что асимптотика решения задачи зависит от количества делителей числа во входе. Вы, возможно, слышали такое утверждение: «количество делителей числа n — это примерно кубический корень из n». Действительно ли у чисел может быть так много делителей?

На самом деле, это, конечно, не так, но оценка в кубический корень дает нужный порядок величин для грубых оценок на числах, с которыми мы имеем дело в реальной жизни (отличие не больше, чем в 4 раза при $n \leqslant 10^{15}$, и не больше, чем в 10 раз при $n \leqslant 10^{18}$). Давайте разберемся, сколько все таки на самом деле делителей у чисел, и как этим пользоваться, а также придумаем новую более точную оценку.

22.2 Субполиномиальность и сверхсоставные числа

Будем обозначать количество делителей числа n как $d(n)$. На самом деле $d(n)$ — это субполиномиальная величина, то есть для достаточно больших n она меньше n^{ε} для сколь угодно малого положительного ε. С доказательством можно, к примеру, ознакомиться, в этой статье. То есть для ооочень больших n можно оценивать $d(n)$ не только как кубический корень, но и как корень четвертой, пятой... любой степени! Однако нас не очень сильно интересуют эти теоретические оценки, потому что по настоящему они достигаются только для невероятно огромных чисел, с которыми мы не работаем. Нас интересуют более практичные оценки.

Максимальное количество делителей неразрывно связано с таким понятием как «сверхсоставные числа» (highly composite numbers). Это такие числа, у которых больше делителей, чем у всех меньших чисел. Первые сверхсоставные числа — это $1, 2, 4, 6, 12, 24\ldots$ Тогда, к примеру, если мы хотим понять, какое максимальное количество делителей есть у чисел, не превосходящих 1000, можно посмотреть в список сверхсоставных чисел, найти там самое больше число, не превосходящее 1000 (это будет 840), и посмотреть, сколько у него делителей (32). На удивление сверхсоставные числа

встречаются не очень часто: существует всего 156 сверхсоставных чисел, не превосходящих 10^{18}. К примеру, сверхсоставные числа можно найти как последовательность на OEIS: oeis.org/A002182.

22.3 Быстрая генерация больших сверхсоставных чисел

Для генерации сверхсоставных чисел есть этот замечательный скрипт. По ссылке есть скрипт, способный очень быстро вычислить все сверхсоставные числа до `MAXN` (при `MAXN`$= 10^{100}$ программа работает меньше секунды), а также ниже представлен список всех всех сверхсоставных чисел до 10^{18} с их количествами делителей и разложениями.

Давайте разберемся, как же этому коду удается так быстро находить такие большие сверхсоставные числа. Для начала придумаем парочку очевидных алгоритмов поиска сверхсоставных чисел. Давайте идти по числам по возрастанию, находить их количества делителей, и если все меньшие числа имели меньше делителей, то новое число будет сверхсоставным. Количество делителей числа можно находить аналогично проверке на простоту за $O(\sqrt{n})$ (все делители разбиваются на пары вида $(x, \frac{n}{x})$, и один из этих делителей будет не больше корня из n). В частности, из этого следует самая простая оценка на количество делителей числа: $d(n) \leqslant 2 \cdot \sqrt{n}$. Тогда мы найдем все сверхсоставные числа до n за $O(n \cdot \sqrt{n})$, что очень долго.

Более умный способ основан на полезной идее о том, что вместо того, чтобы для каждого числа перебирать делители, мы можем действовать наоборот: для каждого числа будем перебирать, какие числа на него делятся. Эта идея похожа на идею решета Эратосфена и, как известно, работает за $O(n \log n)$, потому что числа, делящиеся на k — это k, $2k$, $3k$, … Их $\frac{n}{k}$ штук. Всего мы получим $\frac{n}{1} + \frac{n}{2} + \ldots + \frac{n}{n} = O(n \log n)$ (сумма гармонического ряда). В частности, из этого следует, что в среднем у чисел от 1 до n как раз таки очень мало делителей: $O(\log n)$. Но это все еще долго, такой алгоритм будет работать только для n порядка 10^8.

Для того, чтобы придумать эффективный алгоритм, давайте подробнее изучим нашу задачу. Давайте поймем, чему же равно количество делителей числа n. Пусть $n = p_1^{a_1} \cdot p_2^{a_2} \cdot \ldots \cdot p_k^{a_k}$. Тогда у n есть ровно $(a_1+1) \cdot (a_2+1) \cdot \ldots \cdot (a_k+1)$ делителей, потому что любой делитель n должен состоять только из тех простых, которые есть в n, и при этом степени должны быть не больше, чем степени в n. Поэтому у нас есть a_1+1 вариантов степени p_1 (от 0 до a_1), a_2+1 вариантов степени p_2 и так далее. Мы хотим, чтобы это произведение было как можно больше.

Несложно заметить, глядя на список по ссылке выше, что степени вхождения простых $2, 3, 5, 7\ldots$ в сверхсоставные числа не возрастают. Действительно, если есть, к примеру, число $40 = 2^3 \cdot 3^0 \cdot 5^1$, то можно поменять местами степени тройки и пятерки и получить число $24 = 2^3 \cdot 3^1 \cdot 5^0$, у которого будет ровно столько же делителей, но само число будет меньше.

Собственно, ровно на этой идее и основан алгоритм выше. Он начинает с числа 1 и постепенно генерирует все возможные числа, не превосходящие `MAXN`, у которых степени вхождения простых не возрастают. Так как сумма степеней простых не больше логарифма `MAXN`, таких невозрастающих наборов степеней будет не очень много. После чего, зная разложение, легко посчитать для каждого из них количество делителей. Однако это еще не ответ. Это лишь кандидаты на то, чтобы быть сверхсоставными. Остается отсортировать их и пройтись по порядку, выбирая все числа, у которых больше делителей, чем у всех предыдущих.

22.4 Таблица для повседневного использования

Список сверхсоставных чисел — это, конечно, хорошо, но он не очень удобен для использования на практике. На практике у нас есть некоторое ограничение на n, и мы хотим узнать, какое максимальное количество делителей может быть у числа с такими ограничениями. На этот случай есть такая замечательная табличка:

$\leqslant N$	n	Факторизация	$d(n)$
20	12	$2^2 \cdot 3^1$	6
50	48	$2^4 \cdot 3^1$	10
100	60	$2^2 \cdot 3^1 \cdot 5^1$	12
10^3	840	$2^3 \cdot 3^1 \cdot 5^1 \cdot 7^1$	32
10^4	7560	$2^3 \cdot 3^3 \cdot 5^1 \cdot 7^1$	64
10^5	83 160	$2^3 \cdot 3^3 \cdot 5^1 \cdot 7^1 \cdot 11^1$	128
10^6	720 720	$2^4 \cdot 3^2 \cdot 5^1 \cdot 7^1 \cdot 11^1 \cdot 13^1$	240
10^7	8 648 640	$2^6 \cdot 3^3 \cdot 5^1 \cdot 7^1 \cdot 11^1 \cdot 13^1$	448
10^8	73 513 440	$2^5 \cdot 3^3 \cdot 5^1 \cdot 7^1 \cdot 11^1 \cdot 13^1 \cdot 17^1$	768
10^9	735 134 400	$2^6 \cdot 3^3 \cdot 5^2 \cdot 7^1 \cdot 11^1 \cdot 13^1 \cdot 17^1$	1344
10^{11}	97 772 875 200	$2^6 \cdot 3^3 \cdot 5^2 \cdot 7^2 \cdot 11^1 \cdot 13^1 \cdot 17^1 \cdot 19^1$	4032
10^{12}	963 761 198 400	$2^6 \cdot 3^4 \cdot 5^2 \cdot 7^1 \cdot 11^1 \cdot 13^1 \cdot 17^1 \cdot 19^1 \cdot 23^1$	6720
10^{15}	866 421 317 361 600	$2^6 \cdot 3^4 \cdot 5^2 \cdot 7^1 \cdot 11^1 \cdot 13^1 \cdot 17^1 \cdot 19^1 \cdot 23^1 \cdot 29^1 \cdot 31^1$	26 880
10^{18}	897 612 484 786 617 600	$2^8 \cdot 3^4 \cdot 5^2 \cdot 7^2 \cdot 11^1 \cdot 13^1 \cdot 17^1 \cdot 19^1 \cdot 23^1 \cdot 29^1 \cdot 31^1 \cdot 37^1$	103 680

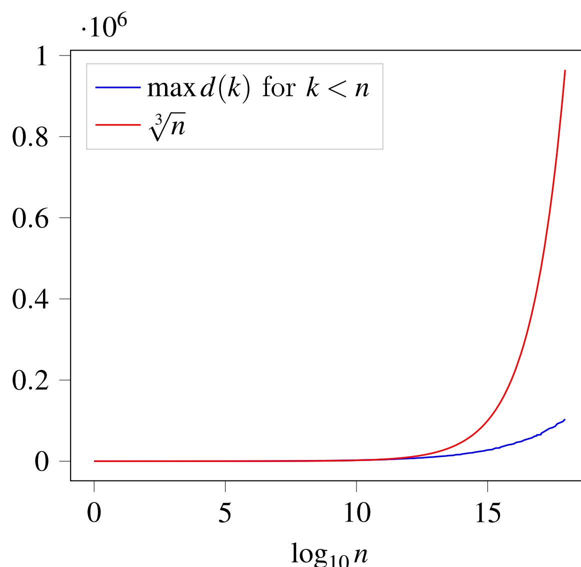
Исходный код таблицы в LaTeX доступен по ссылке

По первому столбцу можно понять ограничения, по второму столбцу можно увидеть наименьшее число с наибольшим количеством делителей, потом его разложение на простые, и последний столбец, наконец, говорит нам то самое количество делителей. Если у вас нет этой таблицы, ее можно, к примеру, получить по двум последовательностям OEIS: oeis.org/A066150, oeis.org/A066151.

22.5 Численный анализ оценки в кубический корень

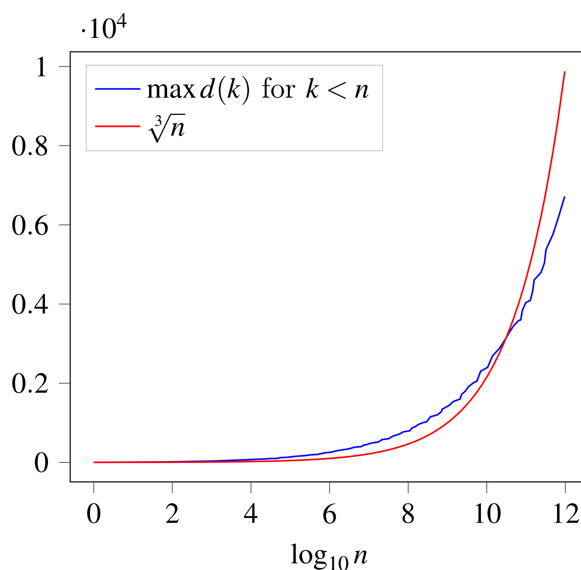
Для точности рекомендуется пользоваться этой таблицей и не полагаться на оценку в кубический корень, однако если этой таблички нет под рукой, то эта оценка даст вам примерное представление о максимальном количестве делителей.

Чтобы лучше понимать эту оценку, давайте проанализируем графики:



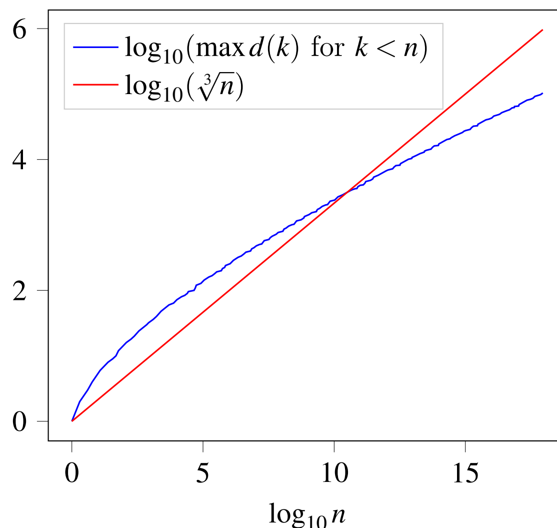
На данном графике на оси X отложены десятичные логарифмы n (то есть количество цифр в числе n), красный график — это кубический корень, а синий график — это максимальное количество делителей среди чисел, меньших n (обратите внимание, что все числа по оси Y умножаются на миллион). Из-за экспоненциального роста не видно, что происходит для чисел, меньших 10^{11}, но и так видно, что для больших n кубический корень уходит сильно выше, но все равно отношение не больше 10 для обозримых чисел.

Давайте ограничим наш график на числа, меньшие 10^{12}, чтобы посмотреть внимательнее на эту часть графика:



Видно, что для маленьких чисел количество делителей все таки больше кубического корня, а где-то в районе между 10^{10} и 10^{11} эти графики меняются местами и уже никогда не встречаются вновь. Как я говорил раньше, сильный всплеск отношения между этими графиками происходит уже на очень больших числах, и для $n \leqslant 10^{15}$ отношение между этими величинами не превосходит 4.

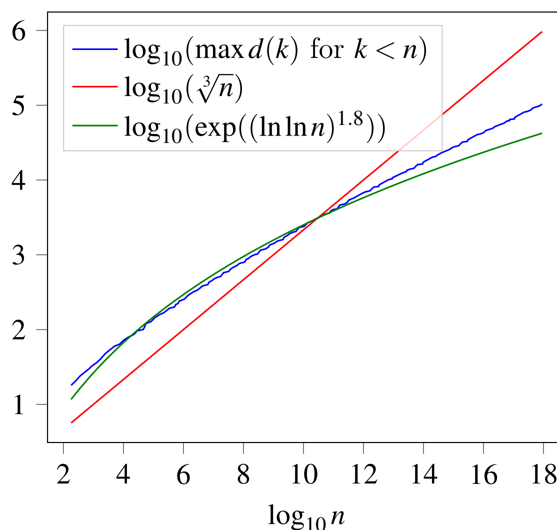
Но если мы смотрим на такие графики, то мы видим только то, что происходит для больших чисел, потому что все маленькие величины становятся просто точкой. Для того, чтобы так не происходило, прологарифмируем также и ось Y:



Теперь происходящее стало еще понятнее.

22.6 Более точная практическая оценка

Как мы уже говорили раньше, оценка в кубический корень очень проста и дает правильный порядок величин, когда нет доступа к таблице. Кроме того, кубический корень достаточно легко вычислить даже без компьютера. Однако если у вас нет таблицы, но при этом есть компьютер, я рекомендую вам другую очень точную оценку на максимальное количество делителей: $\exp((\ln\ln n)^{1.8})$.



Посмотрите, как удивительно точно следуют друг за другом зеленый и синий графики! До 10^{12} они идут настолько ровно рядом, что пропадает проблема с тем, что кубический корень достаточно сильно меньше $d(n)$ для малых n.

Наша новая оценка отличается от реального значения не больше, чем в 1.5 раза для $n \leqslant 10^{14}$, и не больше, чем в 2.5 раза для $n \leqslant 10^{18}$.

Еще одним удивительным фактом является то, что все три графика пересекаются примерно в одном и том же месте между 10^{11} и 10^{12}.

Алгебра

23 Сумма по подмножествам и xor-and-or-свертки 153

23. Сумма по подмножествам и xor-and-or-свертки

Это новая упрощенная и улучшенная версия статьи с меньшим количество математики. Старую версию можно прочитать здесь.

TL;DR SOS-DP — это алгоритм, который за $O(2^k \cdot k)$ находит для каждой из 2^k масок сумму элементов изначального массива по всем подмаскам. Считается при помощи динамики, перебирая биты и постепенно пытаясь их менять. or-xor-and-свертки — это обобщения свертки двух последовательностей через преобразование Фурье, но вместо $c_i = \sum_{j,k: j+k=i} a_j \cdot b_k$ мы сворачиваем $c_i = \sum_{j,k: j \diamondsuit k=i} a_j \cdot b_k$, где $\diamondsuit$ — это какая-то битовая операция (or, xor, and). Все свертки получаются применением какого-то преобразования к исходным последовательностям, поэлементным их перемножением и применением обратного преобразования. Для or-свертки это преобразование — сумма по подмножествам, для and-свертки — сумма по надмножествам, для xor-свертки — преобразование Уолша — Адамара. Во всех случаях, чтобы получить обратное преобразование, необходимо просто обратить все действия, которые делает прямое преобразование, в обратном порядке.

23.1 Сумма по подмножествам

Определение 23.1.1 Пусть дан массив a. Назовем массивом сумм по подмножествам (Sum Over Subsets или SOS-DP) массива a такой массив b, что

$$b_i = \sum_{j: j \subset i} a_j$$

При этом условие $j \subset i$ в данном случае воспринимается как содержание подмножеств единичных битов. То есть если бы мы воспринимали числа как маски. На языке битовых операций это условие можно записать как `i | j = i`, иными словами, j — это число i, в котором занулили некоторые единичные биты.

Вычисление массива сумм по подмножествам является составным блоком для

решения многих задач. К примеру, если элементы массива a — это нули и единицы, обозначающие, является ли данное множество выигрышным, то элемент массива b, соответствующий какому-то множеству, будет положительным тогда и только тогда, когда какое-то его подмножество является выигрышным. Более того, значение в массиве b — это количество выигрышных подмножеств данного множества.

Очевидный алгоритм для поиска массива b, работающий за время $O(n^2)$, — для каждой маски i перебрать все другие маски и просуммировать те, которые являются подмасками i. Если воспользоваться алгоритмом перебора всех подмасок данной маски вместо того, чтобы перебирать все остальные маски в принципе, можно достичь асимптотики $O(3^k) = O(n^{\log_2 3}) \approx O(n^{1.585})$. Однако далее при помощи динамического программирования мы достигнем асимптотики $O(n \log n)$.

Давайте введем динамику `dp`, такую что `dp[mask][ind]` — это сумма по тем подмножествам маски *mask*, в которых изменялись только первые *ind* битов. Если $ind = 0$, то мы не меняли никаких битов, и это просто a_{mask}. Далее при переходе от ind к $ind + 1$ нужно, возможно, поменять текущий бит, либо не менять его. То есть, если бит *ind* в *mask* равен нулю, то мы ничего поменять не можем:

`dp[mask][ind + 1] = dp[mask][ind]`

Если же он равен единице, то мы его можем занулить:

`dp[mask][ind + 1] = dp[mask][ind] + dp[mask ^ (1 << ind)][ind]`

В конце `dp[mask][k]` как раз таки будет равно $b[mask]$, потому что это сумма по подмножествам, где мы можем занулять любые биты. Данную динамику можно слегка упростить, избавившись от второй размерности, и пересчитывать `dp[mask][ind]` в `dp[mask][ind + 1]` in-place. Итоговый алгоритм будет выглядеть следующим образом:

```
1  vector<int> sum_over_subsets(const vector<int>& a) {
2      vector<int> dp = a;
3      for (size_t bit = 1; bit < a.size(); bit <<= 1) {
4          for (size_t mask = 0; mask < a.size(); mask++) {
5              if ((mask & bit) != 0) {
6                  dp[mask] += dp[mask ^ bit];
7              }
8          }
9      }
10     return dp;
11 }
```

Здесь переменная `bit` равна `1 << ind`.

Очевидно, данный код работает за $O(n \cdot k) = O(n \log n)$, потому что первый цикл пробегает по всем k битам, а второй — по всем n маскам.

23.1.1 Обращение суммы по подмножествам

Иногда в задаче бывает нужно делать обратную операцию: по массиву сумм подмножеств b построить изначальный массив a. Есть два подхода к придумыванию решения этой задачи. Можно придумать аналогичную динамику, в которой мы постепенно фиксируем биты. Но можно поступить проще. Можно воспользоваться общей идеей обращения алгоритмов, которая применима в разных контекстах: если у нас уже есть алгоритм, который делает какое-то прямое преобразование, то, чтобы сделать обратное преобразование, достаточно представить, что мы обращаем время вспять, и обратить все действия, которые мы сделали, в обратном порядке. Тогда нетрудно понять, что мы получим следующий алгоритм:

```
vector<int> inverse_sum_over_subsets(const vector<int>& b) {
    vector<int> dp = b;
    for (size_t bit = b.size() >> 1; bit >= 1; bit >>= 1) {
        for (size_t mask = b.size() - 1; mask >= 0; mask--) {
            if ((mask & bit) != 0) {
                dp[mask] -= dp[mask ^ bit];
            }
        }
    }
    return dp;
}
```

Мы обратили порядок обоих циклов, тем самым выполняя все операции в обратном порядке, и если раньше мы что-то прибавили, то, чтобы обратить это действие, достаточно вычесть. Нетрудно заметить, что операции внутреннего цикла на одной итерации внешнего друг от друга не зависят, так что можно перевернуть внутренний цикл обратно. Аналогично, перебор битов во внешнем цикле по возрастанию был произвольным выбором, и для достижения того же результата мы можем перебирать биты в любом порядке, так что на самом деле оба цикла можно перевернуть обратно, и единственным отличием от прямого преобразования будет то, что вместо прибавления мы вычитаем:

```
vector<int> inverse_sum_over_subsets_simplified(const
   vector<int>& b) {
    vector<int> dp = b;
    for (size_t bit = 1; bit < b.size(); bit <<= 1) {
        for (size_t mask = 0; mask < b.size(); mask++) {
            if ((mask & bit) != 0) {
                dp[mask] -= dp[mask ^ bit];
            }
        }
    }
    return dp;
}
```

И теперь уже совсем не составляет труда заметить, что получившийся код, — это в точности то, что мы бы получили, если бы попытались построить аналогичную динамику с последовательным фиксированием битов. Однако идея обращения всех операций в обратном порядке — это максимально общая концепция, которая применима во многих других контекстах.

23.1.2 Сумма по надмножествам

А что, если нам хочется найти не сумму по **под**множествам, а сумму по **над**множествам? Одно можно преобразовать в другое, заменив все нули в масках на единицы, а единицы на нули. Тогда зануление нескольких единиц в новой маске соответствует превращению нескольких нулей в единицы в старой маске, то есть переходу к какой-то надмаске. То есть нам надо просто поменять в алгоритме то, как мы обращаемся с нулями и с единицами. Если раньше, когда в маске какой-то бит был единицей, мы могли поменять его на ноль, то теперь наоборот: если какой-то бит в маске равен нулю, то

мы можем поменять его на единицу. Тем самым, единственное, что нам надо поменять в коде, — это неравенство нулю на равенство:

```
vector<int> sum_over_supersets(const vector<int>& a) {
    vector<int> dp = a;
    for (size_t bit = 1; bit < a.size(); bit <<= 1) {
        for (size_t mask = 0; mask < a.size(); mask++) {
            if ((mask & bit) == 0) {
                dp[mask] += dp[mask ^ bit];
            }
        }
    }
    return dp;
}
```

И обратная операция по получению массива a по массиву b аналогично получается заменой плюса на минус:

```
vector<int> inverse_sum_over_supersets(const vector<int>& b) {
    vector<int> dp = b;
    for (size_t bit = 1; bit < b.size(); bit <<= 1) {
        for (size_t mask = 0; mask < b.size(); mask++) {
            if ((mask & bit) == 0) {
                dp[mask] -= dp[mask ^ bit];
            }
        }
    }
    return dp;
}
```

23.1.3 Сумма по подмножествам для проверки леммы Холла

Лемма Холла — стандартный способ проверки того, что в двудольном графе существует паросочетание, насыщающее левую долю. Она утверждает, что паросочетание есть тогда и только тогда, когда любое подмножество вершин левой доли соединено с не меньшим количество вершин правой доли. В общем случае, если размер левой доли равен n, для проверки потребуется минимум $O(2^n)$ времени, чтобы перебрать все подмножества левой доли. Легко заметить, к примеру, что нас интересуют только в некотором смысле максимальные по включению подмножества вершин левой доли. Иными словами, если к подмножеству вершин левой доли можно добавить еще одну вершину, не увеличив множество их соседей в правой доле, то если лемма Холла не выполнялась, то она и не начнет выполняться для нового подмножества. Таким образом, вместо подмножеств левой доли можно перебирать подмножества правой доли и смотреть на все вершины левой, которые соединены только с какими-то вершинами из этого подмножества. В общем случае это будет работать примерно за $O(2^m)$, где m — количество вершин в правой доле, что совершенно не лучше, чем $O(2^n)$, ведь если $m < n$, паросочетания точно не существует. Однако в конкретных случаях можно сделать проверку сильно быстрее. К примеру, если у каждой вершины графа есть вес w_v, который означает что вершины v есть w_v копий. В таком случае реальное количество вершин в каждой доле равно сумме весов, однако перебирать нам нужно только подмножества типов вершин. В таком случае m может быть сильно меньше n,

и нам остается перебрать все подмножества правой доли и для каждого из них понять, сколько вершин левой доли с ними соединены. Наивный алгоритм сделает это за количество ребер в графе для каждого из 2^m подмножеств, чтобы найти все вершины левой доли, которые соединены только с этим подмножеством, однако при помощи сумм по подмножествам можно сделать это сильно быстрее. Определим `a[mask]` как сумму весов всех вершин левой доли, которые соединены ровно с вершинами правой доли из маски. Такой массив можно посчитать за линейное время от количества ребер в графе. Если мы теперь за $O(2^m \cdot m)$ построим массив `b[mask]`, являющийся суммой по подмножествам массива `a`, то в нем как раз таки и будут храниться суммарные веса всех вершин левой доли, которые соединены с каким-то подмножеством вершин из маски. Теперь лишь остается для каждой маски сравнить это число с суммой весов вершин из маски, и тем самым мы проверим лемму Холла. К примеру, такое решение можно использовать, когда n имеет порядок 10^5, а $m \approx 20$.

23.2 or-свертка и and-свертка

Свертки последовательностей — часто встречающийся шаг в решении задач. К примеру, быстрое преобразование Фурье (знание быстрого преобразования Фурье необязательно для понимания дальнейших алгоритмов) позволяет по двум массивам a и b считать массив c, такой что $c_i = \sum_{j=0}^{i} a_j \cdot b_{i-j}$ за $O(n \log n)$, хотя тривиальный алгоритм сделал бы это только за $O(n^2)$. Такая свертка двух последовательностей бывает очень полезной во многих задачах. Альтернативно можно записать ее так:

$$c_i = \sum_{j,k:j+k=i} a_j \cdot b_k$$

То есть мы суммируем произведения элементов с индексами, в сумме дающими i. Однако по аналогии можно рассмотреть и другую свертку, которая тоже бывает полезна при решении задач: or-свертку. Формула аналогична:

$$c_i = \sum_{j,k:j|k=i} a_j \cdot b_k$$

Найти последовательность c по последовательностям a и b за $O(n \log n)$ нам как раз поможет сумма по подмножествам. Для простоты увеличим длины массивов a и b до ближайшей степени двойки, и тогда мы можем считать, что все три массива имеют длину $n = 2^k$. По аналогии с быстрым преобразованием Фурье, мы сначала преобразуем последовательности a и b в альтернативный вид, в котором для получения свертки достаточно перемножить их поэлементно, а затем сделаем обратное преобразование и тем самым получим последовательность c. Для быстрого преобразования Фурье это преобразование было вычислением значений многочленов в фиксированных точках, а в нашем случае это будет переход от последовательностей к их суммам по подмножествам. За $O(2^k \cdot k) = O(n \log n)$ мы можем построить последовательности f и g, которые являются суммами по подмножествам a и b соответственно. Тогда утверждается, что последовательность h, полученная поэлементным произведением f и g ($h_i = f_i \cdot g_i$) является суммой по подмножествам последовательности c. Почему это так? По определению суммы по подмножествам, мы можем расписать h_i следующим образом:

$$h_i = f_i \cdot g_i = \sum_{j:j \subset i} a_j \cdot \sum_{k:k \subset i} b_k$$

Произведение сумм — это сумма попарных произведений, так что

$$h_i = \sum_{j:j\subset i, k:k\subset i} a_j \cdot b_k$$

То, что j и k являются подмасками маски i эквивалентно тому, что их побитовый or является подмаской i, поэтому эту сумму можно переписать следующим образом:

$$h_i = \sum_{j,k:j|k\subset i} a_j \cdot b_k$$

А это в точности и есть сумма по подмножествам последовательности c, ведь в последовательности c нас интересует сумма произведений по всем парам масок, у которых or равен i, а здесь мы берем сумму произведений по всем парам масок, у которых or — это подмаска i. Иными словами, если мы обозначим $j|k$ за l, мы можем переписать сумму выше как

$$h_i = \sum_{l\subset i} \sum_{j,k:j|k=l} a_j \cdot b_k = \sum_{l\subset i} c_l$$

Таким образом, чтобы из последовательности h получить последовательность c, достаточно применить алгоритм обращения суммы по подмножествам. Итоговый алгоритм or-свертки выглядит следующим образом:

```
1  vector<int> or_convolution(const vector<int>& a, const
      vector<int>& b) { // a.size() == b.size() == 2^k
2      vector<int> f = sum_over_subsets(a);
3      vector<int> g = sum_over_subsets(b);
4      vector<int> h(f.size());
5      for (size_t i = 0; i < f.size(); i++) {
6          h[i] = f[i] * g[i];
7      }
8      vector<int> c = inverse_sum_over_subsets(h);
9      return c;
10 }
```

23.2.1 and-свертка

Легко понять, что для операции and все работает точно так же. Если мы хотим получить свертку

$$c_i = \sum_{j,k:j\&k=i} a_j \cdot b_k$$

достаточно взять сумму по **над**множествам последовательностей a и b, перемножить результаты, а потом взять обращение суммы по надмножествам для этого произведения. Для доказательства можно либо провести абсолютно аналогичные рассуждения, либо вспомнить, что сумма по надмножествам отличалась от суммы по подмножествам заменой в уме всех битов 1 на 0 и наоборот. Операции or и and отличаются друг от друга ровно такой же заменой.

23.3 xor-свертка

23.3.1 Преобразование Уолша — Адамара и его применение

Мы разобрались с or-сверткой и and-сверткой. Остается разобраться с последней битовой операцией — xor. Иными словами, по двум последовательностям a и b мы хотим построить последовательность c, такую что

$$c_i = \sum_{j,k: j \oplus k = i} a_j \cdot b_k = \sum_j a_j \cdot b_{i \oplus j}$$

где $\oplus$ — это операция xor. Для этого нам опять нужно придумать такое преобразование, чтобы поэлементно перемноженные преобразования от a и b дали преобразование от c. Мы определим его следующим образом. Преобразованием последовательности d назовем последовательность e, такую что

$$e_i = \sum_j (-1)^{|i \& j|} d_j$$

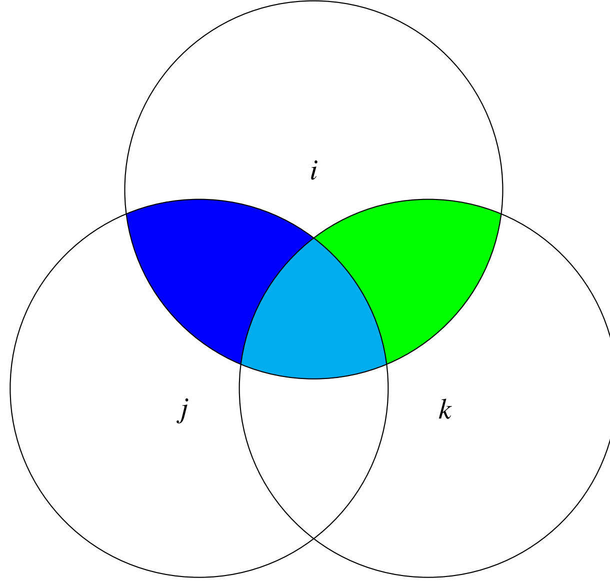
Иными словами, это знакочередующаяся сумма всех элементов d, где маска j берется с плюсом, если их пересечение с маской i четного размера, и с минусом, если нечетного. Такое преобразование называется преобразованием Уолша — Адамара. Почему же такое преобразование подходит? Это следует из того, что четность суммы размеров двух масок равна четности размера их ксора (симметрической разности). Обозначим за f преобразование последовательности a, за g преобразование последовательности b и за h поэлементное произведение f и g ($h_i = f_i \cdot g_i$). Мы хотим доказать, что h — это преобразование Уолша — Адамара от c. Распишем:

$$h_i = f_i \cdot g_i = \sum_j (-1)^{|i \& j|} a_j \cdot \sum_k (-1)^{|i \& k|} b_k$$

Как и в случае с or-сверткой, распишем произведение сумм как сумму попарных произведений:

$$h_i = \sum_{j,k} (-1)^{|i \& j| + |i \& k|} a_j \cdot b_k$$

Число $(-1)^{|i \& j| + |i \& k|}$ зависит от четности $|i \& j| + |i \& k|$. На картинке ниже эта сумма — размер синей области плюс размер зеленой области плюс дважды размер голубой. Но дважды учтенная голубая область не влияет на четность, поэтому четность суммы $|i \& j| + |i \& k|$ равна четности $|i \& (j \oplus k)|$ (размер пересечения i и симметрической разности j и k), потому что это в точности сумма размеров синей и зеленой областей, что легко понять по картинке.



Таким образом, значение h_i можно переписать:

$$h_i = \sum_{j,k} (-1)^{|i\&(j\oplus k)|} a_j \cdot b_k$$

Обозначим $l := j \oplus k$ и просуммируем отдельно по каждому значению l:

$$h_i = \sum_l \sum_{j,k: j\oplus k=l} (-1)^{|i\&l|} a_j \cdot b_k$$

Член $(-1)^{|i\&l|}$ можно вынести из внутренней суммы, после чего внутренняя сумма в точности будет равна c_l по определению:

$$h_i = \sum_l (-1)^{|i\&l|} \sum_{j,k: j\oplus k=l} a_j \cdot b_k = \sum_l (-1)^{|i\&l|} c_l$$

Таким образом, мы получили, что h_i в точности равно преобразованию Уолша — Адамара для последовательности c. Поэтому, чтобы найти последовательность c, нам остается лишь понять, как эффективно производить это преобразование и его обращение.

23.3.2 Вычисление преобразования Уолша — Адамара и его обращения

Мы хотим посчитать по последовательности a такую последовательность f, что

$$f_i = \sum_j (-1)^{|i\&j|} a_j$$

По аналогии с суммой по подмножествам, мы будем считать последовательность f при помощи динамики, постепенно перебирая биты, разрешая им меняться. Введем динамику `dp`, такую что `dp[mask][ind]` — это знакочередующаяся сумма по тем подмножествам маски *mask*, в которых изменялись только первые *ind* битов, и знак определяется размером пересечения масок среди первых *ind* битов. Если $ind = 0$, то мы не меняли никаких битов, и это просто a_{mask}. Далее, при переходе от *ind* к $ind + 1$ нужно, возможно, поменять текущий бит, либо не менять его. Если текущий бит в маске равен нулю, то меняем мы его или нет, знаки остаются теми же самыми, потому что размер пересечения масок не изменился:

`dp[mask][ind + 1] = dp[mask][ind] + dp[mask ^ (1 << ind)][ind]`

Если же текущий бит равен единице, и мы его не меняем, то в пересечение добавляется один новый бит, и знак меняется:

`dp[mask][ind + 1] = dp[mask ^ (1 << ind)][ind] - dp[mask][ind]`

В конце `dp[mask][k]` как раз таки будет равно $f[mask]$, потому что это знакочередующаяся сумма по подмножествам, где мы можем менять любые биты. Данную динамику можно слегка упростить, избавившись от второй размерности, и пересчитывать `dp[mask][ind]` в `dp[mask][ind + 1]` in-place. Однако в данном случае нужно быть осторожным. Два значения, отличающиеся на `(1 << ind)`, в данном случае оба меняются одновременно, так что необходимо обновлять их одномоментно, чтобы значения, через которые мы пересчитываемся, действительно были равны `dp[...][ind]` Итоговый алгоритм будет выглядеть следующим образом:

```
1  vector<int> hadamard_transform(const vector<int>& a) {
2      vector<int> dp = a;
3      for (size_t bit = 1; bit < a.size(); bit <<= 1) {
4          for (size_t mask = 0; mask < a.size(); mask++) {
5              if ((mask & bit) == 0) {
6                  int u = dp[mask], v = dp[mask ^ bit];
7                  dp[mask] = u + v;
8                  dp[mask ^ bit] = u - v;
9              }
10         }
11     }
12     return dp;
13 }
```

Здесь переменная `bit` равна `1 << ind`.

Очевидно, что данный код работает за $O(n \cdot k) = O(n \log n)$, потому что первый цикл пробегает по всем k битам, а второй — по всем n маскам.

Чтобы получить обратное преобразование, можно либо посмотреть на формулу определения преобразования и понять, что если мы применим его к f, то мы получим изначальную последовательность a, в которой все элементы умножены на n, либо же воспользоваться общей стратегией: обратить все действия в обратном порядке. В этом случае это немного сложнее, чем раньше. Перед нами есть значения $x = u + v$ и $y = u - v$, и мы хотим обратно получить u и v. Несложно понять, что $u = (x+y)/2$ и $v = (x-y)/2$. Также нетрудно понять, что циклы (по аналогии с суммой по подмножествам) переворачивать необязательно:

```
1  vector<int> inverse_hadamard_transform(const vector<int>& f) {
2      vector<int> dp = f;
3      for (size_t bit = 1; bit < f.size(); bit <<= 1) {
4          for (size_t mask = 0; mask < f.size(); mask++) {
5              if ((mask & bit) == 0) {
6                  int x = dp[mask], y = dp[mask ^ bit];
7                  dp[mask] = (x + y) / 2;
8                  dp[mask ^ bit] = (x - y) / 2;
9              }
10         }
11     }
12     return dp;
```

```
13 }
```

Обратите внимание, что отличие между прямой и обратной сверткой заключается только в том, что в обратной мы делим пополам. Можно заметить, что мы поделим пополам логарифм раз, так что на самом деле, чтобы сделать обратное преобразование, достаточно сделать прямое, а в конце поделить все элементы на размер массива.

Главное только помнить, что во всех этих преобразованиях можно менять порядки прохода по циклам, но не циклы местами!

Наконец, итоговая xor-свертка будет выглядеть абсолютно аналогично or-свертке:

```
1  vector<int> xor_convolution(const vector<int>& a, const
      vector<int>& b) { // a.size() == b.size() == 2^k
2      vector<int> f = hadamard_transform(a);
3      vector<int> g = hadamard_transform(b);
4      vector<int> h(f.size());
5      for (size_t i = 0; i < f.size(); i++) {
6          h[i] = f[i] * g[i];
7      }
8      vector<int> c = inverse_hadamard_transform(h);
9      return c;
10 }
```

23.3.3 Альтернативный математический взгляд

Если вы знакомы с алгеброй, на xor-свертку можно смотреть иначе. Преобразование Фурье воспринимает входную последовательность как коэффициенты многочлена от одной переменной и считает значения этого многочлена в каких-то точках. Преобразование Уолша — Адамара же воспринимает входную последовательность как коэффициенты многочлена от k переменных, где в каждый одночлен каждая переменная входит в степени 0 или 1 (единичные биты маски в точности и говорят, какие переменные стоят перед этим коэффициентом). К примеру, последовательность a_0, a_1, a_2, a_3 воспринимается как многочлен $P(x_1, x_2) = a_0 + a_1x_1 + a_2x_2 + a_3x_1x_2$. Преобразование Уолша — Адамара в точности находит значения этого многочлена во всех 2^k точках, где все переменные равны ± 1. А так как $x_i^2 = 1$ в таком случае, то произведение двух многочленов P и Q от последовательностей a и b как раз таки дает многочлен R для свертки этих последовательностей c.

Сумму по подмножествам можно альтернативно воспринимать таким же образом. Только считаем мы значения в точках, где значения всех переменных — это 0 или 1. В таком случае выполняется условие $x_i^2 = x_i$. Для and-свертки, как всегда, достаточно поменять нули и единицы местами.

23.4 Источники

- Пункт 15 этого блога рассказывает об альтернативном математическом взгляде на свертки.
- Старая более алгебраическая версия этой главы.

23.5 Задачи для практики

- https://csacademy.com/contest/archive/task/random_nim_generator/
- https://csacademy.com/contest/archive/task/and-closure/

- https://csacademy.com/contest/archive/task/token-grid/
- https://www.codechef.com/NOV19A/problems/MDSWIN

VI Геометрия

24 Поиск пары ближайших точек за $O(n)$ 165

25 Проверка пересечения полуплоскостей на непустоту за $O(n)$ 168

26 Минимальная покрывающая окружность за $O(n)$ 171

27 Поиск пересечения полуплоскостей с точкой внутри 173

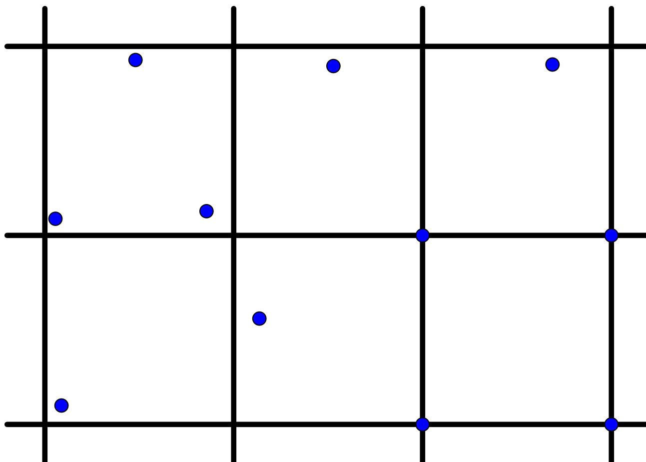
24. Поиск пары ближайших точек за $O(n)$

Решим за линейное время следующую задачу:

Задача 24.1 Даны n точек на плоскости. Требуется найти такую пару точек, что расстояние между ними минимально среди всех расстояний между всеми парами данных точек.

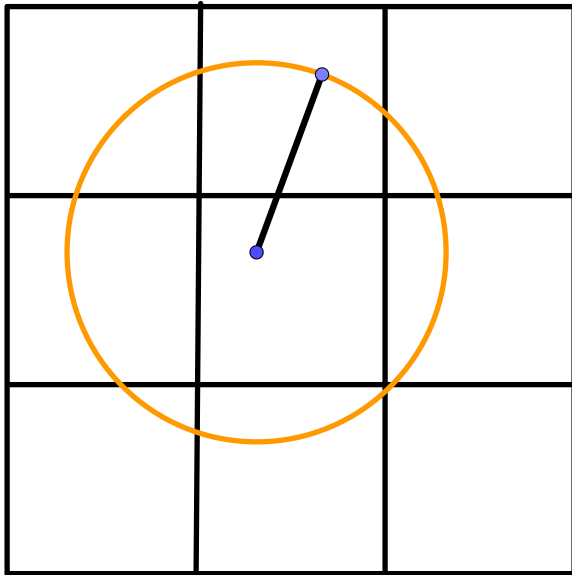
У этой задачи есть детерминированный алгоритм, основанный на идее «разделяй и властвуй», работающий за $O(n \log n)$. Мы же рассмотрим рандомизированный алгоритм, работающий за ожидаемое время $O(n)$.

Давайте будем постепенно добавлять точки по одной в случайном порядке, поддерживая пару ближайших точек. Пусть на данный момент уже было добавлено сколько-то точек, и текущее минимальное расстояние равно d. Побьем всю плоскость на квадраты $d \times d$ и для каждого квадрата в хеш-таблице будем хранить все точки, находящиеся в этом квадрате. При этом можно заметить, что так как на данный момент минимальное расстояние между точками равно d, то расстояние между любой парой точек не меньше d, поэтому в каждом квадрате не больше 4 точек.



Заметим, что если мы добавляем какую-то новую точку, то если минимальное

расстояние изменилось, то есть какая-то точка, которая находится на расстоянии меньше d от новой точки. При этом обратим внимание на то, что эта точка должна находиться либо в том же квадрате $d \times d$, либо в соседнем (по стороне или диагонали), ведь если нарисовать круг радиуса d с центром в новой точке, он целиком будет лежать в квадрате $3d \times 3d$, в центральном квадрате которого лежит новая точка.



Так как в каждом квадрате лежит не больше 4 точек, нам придется перебрать не более $9 \cdot 4 = 36$ точек в поиске пары для новой точки. То есть добавление новой точки будет происходить за константное время. Однако если мы все таки нашли новую пару, то нам придется перестроить всю структуру квадратов, потому что d уменьшится.

Кажется, что такой алгоритм будет работать за квадратичное время, однако вспомним, что мы добавляем точки в случайном порядке! Заметим, что если мы перестроили структуру после добавления k-й точки, то это одна из двух точек в паре самых близких. Так как эта точка была выбрана случайно, то вероятность такого события — $\frac{2}{k}$ (если есть несколько одинаковых минимальных расстояний, то либо до добавления этой точки расстояние уже было таким, и ничего перестраивать не надо, либо же во всех таких минимальных расстояниях одной из точек является наша новая точка, и тогда вероятность еще меньше — $\frac{1}{k}$). Если это событие произошло, то нам нужно перестроить заново структуру. Это происходит за $O(k)$. Поэтому ожидаемое время работы равно $\sum_{k=1}^{n} \frac{2}{k} \cdot k = \sum_{k=1}^{n} 2 = 2n = O(n)$. Что и требовалось доказать.

Замечание 24.0.1 Если координаты точек целые, то в алгоритме можно обойтись без вещественных чисел. Вместо минимального расстояния будем хранить его квадрат, а делить на квадраты будем со стороной не d, а $\lfloor d \rfloor$. Очевидно, от этого асимптотика не поменяется.

Замечание 24.0.2 Чтобы получить номер квадрата, в котором находится точка, вы скорее всего будете делить его координаты на d. Будьте внимательны, что если в множестве есть совпадающие точки, то d может стать равно нулю. В этом случае нужно сразу завершиться, потому что более близких точек точно уже не будет.

С реализацией можно ознакомиться по ссылке.

Замечание 24.0.3 Легко обобщить этот алгоритм для бо́льших размерностей. Нужно побить пространство на кубы со стороной d и искать пару для новой точки во всех соседних кубах. Для фиксированной размерности алгоритм будет все еще линейным.

24.1 Задачи для практики

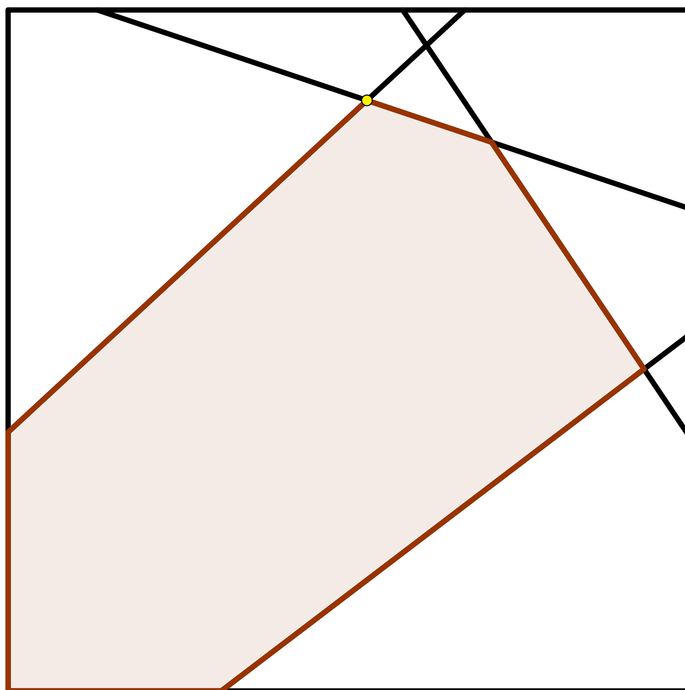
- https://codeforces.com/contest/120/problem/J

25. Проверка пересечения полуплоскостей на непустоту за $O(n)$

Решим за линейное время следующую задачу:
Задача 25.1 Даны n полуплоскостей. Требуется найти точку, лежащую во всех этих полуплоскостях, либо сказать, что такой нет.

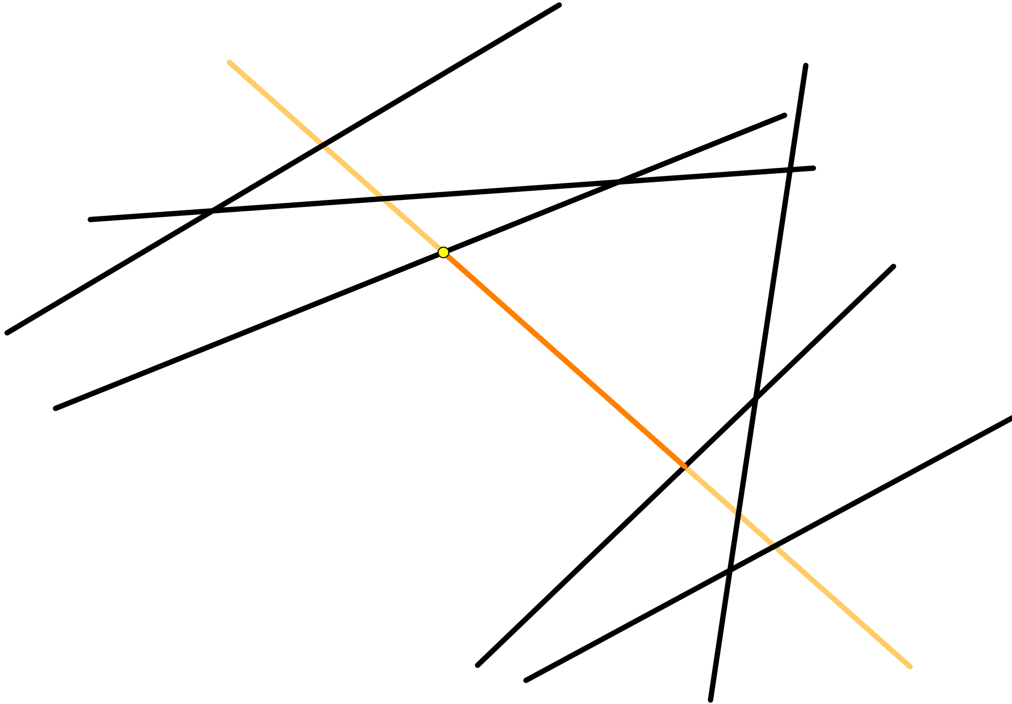
Задачу пересечения полуплоскостей можно решить за время $O(n\log n)$ различными способами. Они находят не только одну точку, но все множество пересечения. Мы же рассмотрим рандомизированный алгоритм, работающий за ожидаемое время $O(n)$.

Давайте будем добавлять полуплоскости по одной в случайном порядке, поддерживая самую высокую точку в множестве пересечения уже добавленных полуплоскостей (Если таких точек несколько, то самую левую из них. При этом изначально ограничим все квадратом $[-10^9, 10^9] \times [-10^9, 10^9]$ при помощи четырех полуплоскостей, чтобы не было проблем с тем, что эта точка бесконечно удалена).



Тогда если при добавлении новой полуплоскости эта точка лежит в очередной полуплоскости, то ничего не поменяется, потому что множество пересечения могло только уменьшиться, но при этом его верхняя точка сохранилась.

Если же эта точка не лежит в новой полуплоскости, то придется заново искать самую верхнюю точку. Заметим, что эта новая точка обязана лежать на прямой, высекающей новую полуплоскость. То есть нам надо найти самую высокую точку на прямой, лежащую в пересечении полуплоскостей. Но ведь все остальные полуплоскости высекают на этой прямой какие-то лучи. Поэтому нам надо найти самую высокую точку в пересечении лучей. Для этого надо отдельно посмотреть на все лучи, смотрящие вниз, и взять из них самый нижний (начинается в точке A). Аналогично взять все лучи, смотрящие вверх, и взять из них самый верхний. Проверить, что эти два луча пересекаются, и в таком случае взять точку A. Это можно сделать за линейное время.



Почему этот алгоритм работает за линейное время? Доказывается это точно так же, как и для поиска двух ближайших точек на плоскости. Самая верхняя левая точка лежит на пересечении двух прямых, высекающих полуплоскости. Тогда вероятность того, что случайно выбранная прямая лежит на краю равна $\frac{2}{k}$. Асимптотика алгоритма равна $\sum_{k=1}^{n} k \cdot \frac{2}{k} = \sum_{k=1}^{n} 2 = 2n = O(n)$.

Замечание 25.0.1 Если больше, чем две полуплоскости, пересекаются в одной точке, то легко заметить, что вероятность от этого становится только меньше. Либо эта точка и раньше уже была самой верхней левой, либо же вероятность правильно выбрать нужную прямую не больше $\frac{2}{k}$.

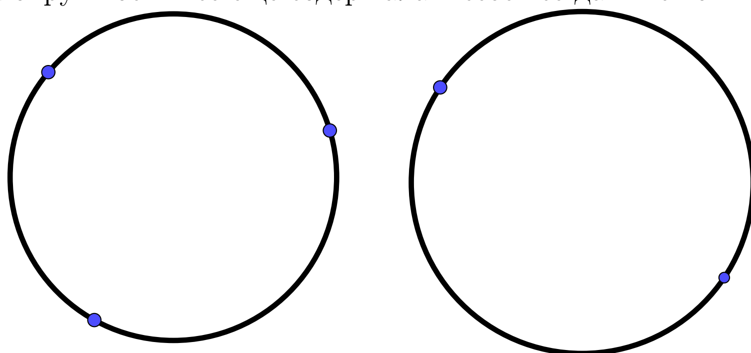
26. Минимальная покрывающая окружность за $O(n)$

Решим за линейное время следующую задачу:

Задача 26.1 Даны n точек на плоскости. Требуется найти окружность минимального радиуса, такую что все данные точки лежат внутри или на границе этой окружности.

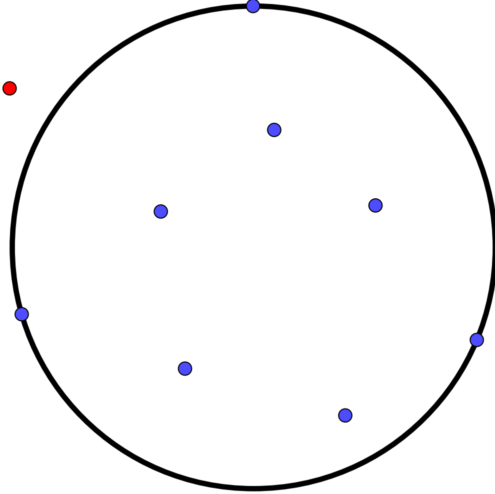
В отличие от других рассмотренных рандомизированных геометрических алгоритмов, у которых есть общеизвестные детерминированные аналоги, работающие в $\log n$ раз дольше, в данном случае рандомизированное решение будет единственным применимым с практической точки зрения.

Есть два варианта того, как будет выглядеть минимальная покрывающая окружность: либо это описанная окружность какой-то тройки из данных точек, либо окружность, построенная на отрезке между двумя какими-то точками как на диаметре, потому что во всех других случаях ее радиус можно немного уменьшить так, чтобы эта окружность все еще содержала в себе все данные точки.



Давайте по очереди добавлять точки в случайном порядке, поддерживая минимальную покрывающую окружность текущего набора точек. Если новая точка лежит внутри текущей окружности, то ничего менять не надо, а в противном случае нужно перестроить окружность. Стоит заметить, что если новая точка не лежит в старой окружности, то она обязана лежать на новой минимальной окружности. На окружности лежит 3 точки, поэтому вероятность такого события — $\frac{3}{k}$, где k — индекс новой точки (если на окружности лежит больше 3 точек, то эта окружность была минимальной еще

до добавления текущей точки, а если окружность построена на двух точках как на диаметре, то вероятность еще меньше — $\frac{2}{k}$). Теперь если мы построим линейный алгоритм, который находит минимальную покрывающую окружность, проходящую через новую точку, то итоговая асимптотика алгоритма будет $\sum_{k=1}^{n} \frac{3}{k} \cdot k = \sum_{k=1}^{n} 3 = 3n = O(n)$.



Алгоритм нахождения минимальной покрывающей окружности, проходящей через новую точку аналогичен алгоритму, который мы только что рассмотрели. Мы перебираем все остальные точки в случайном порядке, и если новая точка лежит вне текущей окружности, то эту окружность нужно перестроить. Вероятность того, что новая точка лежит на минимальной окружности не больше $\frac{2}{k}$, поэтому алгоритм будет линейным.

Остается последний этап: зафиксированы уже две точки и нужно построить минимальную покрывающую окружность, проходящую через них. Алгоритм аналогичен. Начинаем с окружности, построенной на отрезке между этими двумя точками как на диаметре, а затем добавляем остальные точки в случайном порядке. Вероятность, что окружность нужно перестроить не больше $\frac{1}{k}$, при этом если зафиксированы уже три точки на окружности, то окружность определяется однозначно.

Получили трехслойный алгоритм, каждый из которых, как ни удивительно, работает за $O(n)$.

27. Поиск пересечения полуплоскостей с точкой внутри

Алгоритмы нахождения пересечения полуплоскостей достаточно сложные, поэтому стоит пользоваться ситуациями, когда задачу нужно решать не в общем случае. К примеру, если все полуплоскости «смотрят вниз», то это уже задача Convex Hull Trick. Если нужно лишь проверить пересечение полуплоскостей на непустоту, то это можно сделать при помощи линейного рандомизированного алгоритма [1]. В этой главе же мы рассмотрим ситуацию, когда нам известно, что какая-то точка P обязательно лежит строго внутри пересечения полуплоскостей.

Замечание 27.0.1 Обратите внимание, что точка должна лежать строго внутри, поэтому нельзя просто сначала найти точку на границе при помощи рандомизированного линейного алгоритма, а потом запустить алгоритм из этой главы.

Пусть точка P лежит в пересечении полуплоскостей. Тогда давайте сначала переместим точку P в начало координат. Все полуплоскости тоже сдвинем на вектор $-P$. Теперь начало координат лежит в пересечении полуплоскостей. Давайте все прямые вида $ax+by+c=0$ заменим на точки $\left(\frac{a}{c}, \frac{b}{c}\right)$. Теперь мы перешли к двойственной задаче, и у полученного набора точек нужно найти выпуклую оболочку, что уже намного проще. После этого мы найдем список индексов прямых, которые образуют пересечение полуплоскостей.

[1] 25

VII C++, среды, стрессы, тактика и стратегия

28 Генерация случайных чисел 176
29 Стресс-тестирование 181
30 Быстрый ввод-вывод в C++ 187

Эта глава посвящена окружению, с которым вы работаете, а не алгоритмам. Как максимально полно использовать возможности C++ для спортивного программирования, как правильно генерировать случайные числа, какой средой программирования стоит пользоваться, как писать стресс-тесты, как готовиться к олимпиадам, какой стратегии придерживаться на самой олимпиаде, и так далее.

28. Генерация случайных чисел

Часто бывает так, что ваши решения задач зависят от случайных чисел. Стандартным примером будет являться декартово дерево, в котором логарифмическая высота достигается в том случае, если потенциалы будут выбраны случайно. Когда ваша программа использует случайные числа, нужно быть внимательным, чтобы не попасть в какую-нибудь ловушку. В этом разделе мы как раз поговорим про то, какие бывают ловушки, как в них не попасться, а также как упростить себе жизнь. В основном речь будет идти про C++, однако некоторые вещи можно по аналогии перенести в другие языки.

28.1 mt19937

Первое, что нужно сделать, когда вы работаете со случайными числами, — это забыть про функцию `rand`. Не стоит использовать ее ни-ко-гда! И на это есть три причины.

Первая причина заключается в том, что случайные числа, которые генерирует `rand`, — это «плохие» случайные числа. Дело в том, что компьютер не может сгенерировать по настоящему случайные числа. Поэтому вместо этого он генерирует «псевдослучайные» числа. Существуют разные способы генерации псевдослучайных чисел, и способ, который используется в `rand` далеко не самый лучший. В них легко можно вычленить периодичности и т.п.

Вторая причина более обозрима. Она заключается в том, что эта функция платформозависима. По стандарту она генерирует случайное число от нуля до `RAND_MAX`. Проблема заключается в том, что в linux это число `RAND_MAX` совпадает с максимальным числом, которое может храниться в типе `int` ($2^{31}-1$), и все хорошо. Однако в windows `RAND_MAX` равен 32767 ($2^{15}-1$), а это на самом деле очень маленькое число. Если вы не знаете этого и хотите генерировать много случайных чисел, они будут очень часто повторяться в таком случае. Другая проблема встает тогда, когда вам нужно генерировать действительно большие числа. В этом случае с функцией `rand` придется изворачиваться и использовать конструкции типа `RAND_MAX * rand() + rand()`.

И самое ужасное произойдет в тот момент, когда вы захотите использовать функцию

`random_shuffle`, которая случайным образом перемешивает элементы массива. Для ее работы нужна генерация случайных индексов массива, то есть случайных чисел от 0 до $n-1$, однако число n вполне вероятно может быть сильно больше, чем 32767. В таком случае сгенерированная перестановка абсолютно не является случайной. К примеру, если $n = 3 \cdot 10^6$, тесты показывают, что каждый элемент находится в среднем на расстоянии $2\%n$ позиций от своего изначального места, хотя с теоретической точки зрения это должно быть 33%.

> **Замечание 28.1.1** Начиная с C++17 функция `random_shuffle` больше не поддерживается.

Третья причина заключается в том, что генератор случайных чисел, который мы рассмотрим далее работает просто быстрее, чем `rand`. Подробнее мы поговорим про это позже.

Для решения всех этих проблем подойдет генератор случайных чисел `mt19937`, добавленный в C++11. Он вне зависимости от компилятора генерирует случайные числа от 0 до $2^{32}-1$ (обратите внимание, что здесь генерируется случайное `unsigned int` число). Этот генератор основан на простом числе Мерсена $2^{19937}-1$ (его название — это как раз сокращение от его параметров: «Mersenne Twister pseudo-random generator of 32-bit numbers with a state size of 19937 bits»). Такой генератор намного более «рандомный» и его период — это как раз $2^{19937}-1$, что является невероятно большим числом (примерно 10^{6000}). Давайте рассмотрим пример работы этого генератора:

```
1 mt19937 rnd(4321);
2 cout << rnd() << ' ' << rnd() << endl;
```

Данный код создает генератор под названием `rnd` с начальным сидом 4321. После этого генератор можно использовать как функцию. Данный код выведет на экран два случайных числа от 0 до $2^{32}-1$. Стоит обращать внимание на то, что если присвоить значение `rnd()` переменной типа `int`, то значение может быть отрицательным. Однако если вы сразу знаете, что вам нужно число из какого-то диапазона, то взяв значение по модулю вы избежите отрицательных чисел:

`rnd() % 1000`

Кроме того, этот генератор можно использовать при перемешивании элементов массива. Для этого нужно воспользоваться функцией `std::shuffle`, аналогичной `random_shuffle`. Она принимает начало и конец последовательности, а также генератор. К примеру:

`shuffle(a.begin(), a.end(), rnd);`

Также стоит обратить внимание на то, что мы не только выигрываем в качестве генератора и величине генерируемых чисел, но и во времени. Генерация числа при помощи `rnd` в 3 раза быстрее генерации при помощи `rand`.

Если вам нужно генерировать еще бóльшие числа, у `mt19937` есть старший брат `mt19937_64`, который генерирует уже 64-битные случайные числа.

Стоит не забывать и о платформенной независимости. `mt19937` не только генерирует большие случайные числа на любой платформе, он кроме того генерирует одни и те же числа (при фиксированном сиде) на любой платформе. Так что вы можете быть уверены, что когда вы засылаете ваше решение в систему, оно будет там работать точно так же, как и на вашем компьютере.

Еще одним плюсом может быть то, что вы можете создавать несколько разных `mt19937` генераторов в одной программе (с одним и тем же сидом, либо с разными) и использовать их независимо. Как это можно использовать, каждый решит для себя

сам. К примеру, если вы делаете in-code стресс, вы можете не передавать решениям входные данные, сгенерированные случайно, а генерировать их прямо по ходу решения, и если вы используете один и тот же сид, но разные генераторы в двух решениях, они будут генерировать одни и те же числа.

28.2 uniform_int_distrubition

В C++ помимо mt19937 есть большое количество удобных вспомогательных инструментов для работы со случайными числами.

Первый пример, который мы рассмотрим, — это uniform_int_distribution. Это инструмент, позволяющий генерировать случайное целое число в заданном диапазоне. Легче всего понять его принцип работы на примере:

```
1 mt19937 rnd(4321);
2 uniform_int_distribution<int> distrib(1, 10);
3 cout << distrib(rnd) << endl;
```

Мы создаем объект класса uniform_int_distribution. В шаблон мы передаем, какого типа должны возвращаться числа, в данном случае это int (его можно опускать, потому что компилятор сам догадается о типе из типа левой и правой границы). Затем в конструктор передается два числа — левая и правая граница отрезка, в котором будут генерироваться числа. После чего для того, чтобы сгенерировать случайное число, нужно передать в uniform_int_distribution наш генератор случайных чисел. В данном случае это mt19937, который мы заранее определили. Такой код выведет случайное целое число от 1 до 10 включительно.

Стандартным способом сгенерировать число в полуинтервале от l до r был бы следующий код:

```
1 int uniform_distribution(int l, int r) {
2     return rnd() % (r - l) + l;
3 }
```

В случае, если нужно сгенерировать случайное число от 0 до r, код можно упростить, просто взяв значение rnd() по модулю r. uniform_int_distribution не только упрощает этот процесс, но и спасает вас от неожиданных проблем.

Обратите внимание, что если rnd генерирует случайное число от 0 до C, то rnd() % r — это случайное число от 0 до r только в том случае, если C делится на r. В противном случае остатки 0, 1, ..., $(C-1) \bmod r$ будут генерироваться немного чаще, чем все остальные. Вы вряд ли как-то это сможете заметить, если r — это, к примеру, 3, однако для больших r это может привести к неприятным последствиям. Пускай $r = \lfloor \frac{2 \cdot C}{3} \rfloor$. Тогда для остатков от 0 до $\frac{r}{2}$ есть два возможных числа с такими остатками, а для остатков, больших $\frac{r}{2}$, таких чисел по одной штуке. Это значит, что маленькие остатки будут генерироваться в среднем в два раза чаще, чем большие. uniform_int_distribution как раз таки помогает решить эту проблему.

Момент, который стоит все таки отметить — это то, что в отличие от mt19937, uniform_int_distribution все таки платформозависим. То есть при фиксированном сиде mt19937 могут выдаваться разные числа на разных компиляторах. Не то, чтобы это было очень серьезной проблемой, но стоит учитывать, что у вас на компьютере и в тестирующей системе могут генерироваться разные числа.

28.3 Как генерировать случайные числа по модулю

Сделаем небольшое отступление. Как мы поняли, если нам нужно генерировать случайное число из какого-то отрезка, нам в этом поможет `uniform_int_distribution`. Однако что бы мы делали, если бы у нас его не было? Пришлось бы использовать плохой генератор, который генерирует числа неравномерно? Давайте приведем рандомизированный алгоритм, который за ожидаемое время $O(1)$ вернет нам случайный равномерно распределенный остаток по модулю r.

Как мы уже поняли, если 2^{32} делится на r, то все хорошо. Мы должны просто взять остаток от деления `rnd()` на r. Однако если r не является степенью двойки, распределение такого остатка будет неравномерным. Давайте избавимся от этих последних остатков, которые мешают равномерности. То есть возьмем число X, равное $2^{32} - (2^{32} \bmod r)$. Такое число будет делиться на r, но при этом будет не меньше 2^{31} (если $r \geqslant 2^{31}$, то такое число не меньше r, которое не меньше 2^{31}, а если $r < 2^{31}$, то мы вычтем что-то меньшее 2^{31}, поэтому получим число, которое не меньше 2^{31}). Тогда давайте сделаем так: если случайно сгенерированное число q меньше X, то вернем $q \bmod r$, что будет равномерно распределенным случайным остатком, а если q не меньше X, то повторим генерацию заново. Так как X не меньше $\frac{2^{32}}{2}$, то вероятность успеха каждый раз не меньше $\frac{1}{2}$, поэтому нам в среднем понадобится сгенерировать не больше двух случайных чисел.

28.4 Другие распределения

По аналогии с `uniform_int_distribution` в C++ есть много других распределений. К примеру, `normal_distribution` и `exponential_distribution`, которые соответствуют нормальному и экспоненциальному распределениям. Маловероятно, что это может понадобиться вам при решении задач, но все же.

А вот что действительно может понадобиться вам, так это генерировать случайное вещественное число. Для этого подойдет `uniform_real_distribution`. Его использование аналогично `uniform_int_distribution`, но только теперь генерируется не случайное целое число из отрезка, а случайное вещественное.

28.5 Выбор сида рандома

Важным вопросом является выбор сида рандома. Формально есть всего два варианта выбора сида: детерминированный (фиксированное число) и случайный. Давайте поймем, какой вариант нужно использовать в какой ситуации.

Чаще всего (частота употребления — это, конечно, индивидуальная вещь, но все же) вы хотите использовать константный сид рандома (мы ранее использовали всегда число 4321, но это может быть ваше любимое число). Какие у этого плюсы? В таком случае ваши случайные числа весьма условно случайны. На самом деле они вполне детерминированы. Каждый раз, когда вы запускаете программу, вам выдаются одни и те же числа. Чем же это хорошо? Тем, что вы контролируете ситуацию.

Если ваша программа не работает, и вы пытаетесь найти ошибку, вы знаете, что при каждом запуске программе даются одни и те же случайные числа, и вы не столкнетесь с тем, что один раз ошибка была, потом пропала, а воспроизвести ее не получается.

Другой плюс — это уверенность в результате. Вы знаете, что когда вы зашлете решение в систему, оно будет работать там на тестах точно так же, как и на вашем компьютере. Кроме того, если вы получили по задаче AC, то вы уверены, что сколько бы раз не происходило перетестирование, ваше решение все равно будет проходить тесты.

Если же вы используете случайный сид, то при каждом запуске числа, которые выдает ваш случайный генератор, меняются. В результате чего может сложиться такая ситуация, что после окончания контеста все решения будут перетестированы, и вам не повезет со случайными числами, в результате чего ваше решение не пройдет тесты.

В каком же случае все таки нужно использовать случайный сид (про то, как сгенерировать случайный сид, мы поговорим позже)? Во-первых, в любых контестах со взломами (codeforces, topcoder и т.п.). Если в таких соревнованиях ваш сид не случайный, то человек, который смотрит на ваш код, может запустить у себя локально ваше решение, посмотреть, какие случайные числа оно генерирует, и без труда построить контртест, тем самым вся ваша случайность будет абсолютно бесполезна.

Во-вторых, это может понадобиться вам, когда вы хотите запустить ваше решение несколько раз и проверить, что от изменения сида вывод не меняется. Это можно сделать, меняя каждый раз сид руками в коде, но после этого каждый раз придется перекомпилировать программу. Случайный сид может быть хорошим решением этой проблемы.

В-третьих, если вы пишете стресс тесты (29). Есть два подхода в этом плане. Первый — передавать сид как аргумент командной строки, либо выбирать его случайно. Если вы все таки решили выбрать сид случайно, то вам очень важно «качество» этого случайного сида. Об этом мы как раз и поговорим далее.

Случайным сидом обычно выбирают текущее время, потому что это некоторая меняющаяся величина. Самый простой способ получения времени — это `time(0)`. Эта функция возвращает текущее время в секундах. Этот вариант подойдет вам, если вы хотите запустить ваше решение несколько раз и посмотреть, что оно делает при разных сидах (если вы запускаете ваше решение не чаще, чем раз в секунду). Однако эту функцию категорически не стоит использовать в двух других случаях.

В случае стресс тестирования вы хотите прогонять тысячи тестов в секунду, но при использовании `time(0)` ваш генератор целую секунду будет генерировать вам один и тот же тест, поэтому поиск неправильного теста замедляется в тысячи раз.

В случае взломов проблема не так очевидна. На самом деле человек, который вас взламывает, может подгадать, в какую секунду ваше решение будет тестироваться и подготовить тест специально для нее. Если этот вариант кажется вам невозможным, то есть и более реалистичный случай. Можно сделать тест, который является контртестом сразу к 60 разным сидам. И если эти сиды — это последовательные секунды, то здесь для взлома остается лишь совершить его в правильную минуту, что совершенно не является проблемой.

Что же делать? Нужно использовать время в наносекундах (миллиардная доля секунды). Тогда подгадать сид точно уже не представляется возможным. Получить текущее время в наносекундах можно при помощи следующей строки:

`chrono::steady_clock::now().time_since_epoch().count()`

И это можно использовать как сид рандома:

`mt19937 rnd(chrono::steady_clock::now().time_since_epoch().count());`

Есть еще способ через `random_device`. Это генератор случайных чисел, который недетерминирован сам по себе:

```
1 random_device rd;
2 mt19937 rnd(rd());
```

Однако, к сожалению, это тоже платформозависимое решение. На windows `random_device` все таки детерминирован, поэтому этот способ не рекомендуется использовать.

29. Стресс-тестирование

29.1 Общие концепции

Стресс-тестирование — это не тестирование задачи в последние минуты контеста под большим давлением. Стресс-тестирование — это тестирование вашего кода сразу на большом количестве автоматически генерируемых тестов для поиска того теста, на котором ваше решение работает неправильно.

Стандартный сценарий, когда вам нужен стресс: вы заслали решение в систему, получили WA, но ни ошибка в коде, ни неработающий тест никак не хотят находиться. Для стресс-тестирования вам нужны следующие компоненты:

1. smart.cpp. Ваше решение, которое по какой-то причине не проходит тесты.
2. stupid.cpp. Тупое решение. Чаще всего за большую асимптотику. Очень важно, чтобы это решение было максимально тупым, и вы были уверены, что в нем нет никаких ошибок.
3. gen.cpp. Генератор тестов. Чаще всего тесты генерируются случайные, при этом они должны быть не очень большие, чтобы тупое решение тоже могло быстро на них работать, а также, чтобы вы могли потом найти ошибку в своем коде, используя этот тест. Когда вы нашли неработающий тест, попытайтесь максимально его минимизировать. Уменьшить константу в генераторе и так далее. Чем проще тест, тем легче вам будет исправлять ошибку.
4. stress.sh. Скрипт, который будет повторять следующий процесс: сгенерировать тест, запустить на нем умное решение, запустить на нем тупое решение, сравнить ответы. Если ответы равны, нужно повторить процесс, а если не равны, выдать тот тест, на котором ваше решение работает неправильно, и завершиться.
5. checker.cpp. Пункт, который редко будет вам нужен, но все же может понадобиться, — это чекер. Не всегда в задаче существует единственный правильный ответ на каждый тест. К примеру, если задача заключается в том, чтобы разложить число на два множителя, то разных ответов может быть несколько, и каждый из них при этом верный. Поэтому вам может понадобиться написать программу, которая проверяет правильность вашего ответа, учитывая ответ тупого решения.

> Также бывает так, что тупого решения написать нет возможности, потому что любое тупое решение по сложности сопоставимо с вашим умным решением. В этом случае вам может помочь некоторая промежуточная мера, когда у вас есть только чекер. Этот чекер может сделать некоторые проверки ответа на адекватность (к примеру, проверить, что в задаче о разложении на два множителя, произведение этих множителей равно входному числу). Эта проверка может не покрывать все случаи, когда ваше решение работает некорректно, но все равно может помочь вам найти нужный тест.

smart.cpp у вас уже есть, потому что это ваше решение. Чтобы написать stupid.cpp, надо просто представить, что ограничения в задаче маленькие, и написать другое решение. Как уже было сказано ранее, очень важно писать максимально простой код, чтобы не допускать ошибок. checker.cpp принимает на вход три файла: входной файл, правильный ответ и ваш ответ, и проверяет по ним корректность вашего ответа.

Теперь осталось разобраться, как правильно писать gen.cpp и stress.sh.

> **Замечание 29.1.1** Обратите внимание, что здесь речь идет про shell стрессы в linux. В других случаях могут отличаться детали, однако пока мы говорим больше про общие концепции. Конкретные несоответствия будут освещены в соответствующих секциях позже.

Генератор должен создавать случайный тест (небольшого размера). Рекомендуется использовать фиксированный сид рандома, чтобы тесты были воспроизводимы. При этом мы хотим запускать генератор много раз и получать разные тесты. Давайте для этого передавать сид рандома в аргументах командной строки. Функция `main` может принимать два параметра: `agrc` и `argv`. `argc` — это количество аргументов командной строки, а `argv` — это массив аргументов размера `argc`. К примеру, если вы скомпилировали ваш код в файл `main` и запускаете его командой `./main aba 123 8`, то `argc` будет равен 4, а в `argv` будут храниться `./main`, `aba`, `123` и `8`.

Нам нужно передавать всего один параметр, который будет храниться в `argv[1]`. Правда, он там будет храниться в типе `char*` (строки из C), а нам нужно число. Для перевода можно воспользоваться функцией `atoi`.

Кроме того, стоит помнить о том, что пользоваться стоит не функцией `rand`, а генератором `mt_19937`.

Если вы все таки решили не использовать аргументы командной строки, вам нужен случайный сид. Обычно для этого используют текущее время. Однако вам нужно, чтобы это текущее время менялось часто, а не каждую секунду, к примеру. Потому что в таком случае вы целую секунду будете кучу раз запускать один и тот же тест. Для этого вам поможет следующая инициализация рандома:

`mt19937 rnd(chrono::steady_clock::now().time_since_epoch().count());`

Подробнее про генерацию случайных чисел можно прочитать в главе 28.

Вооружившись всеми этими знаниями, давайте напишем генератор для задачи «A+B».

```
1 #include <iostream>
2 #include <random>
3 #include <cstdlib>
4
5 using namespace std;
6
7 int main(int argc, char* argv[]) {
8     int seed = atoi(argv[1]);
```

```
    mt19937 rnd(seed);

    int a = rnd() % 1000;
    int b = rnd() % 1000;

    cout << a << ' ' << b << endl;
    return 0;
}
```

Теперь наша задача заключается в том, чтобы по очереди запускать `gen`, записывать его результат в файл, запускать два решения и сравнивать их ответы. Рассмотрим разные варианты того, как это можно делать.

29.2 Linux: Shell стрессы

Если вы пользуетесь linux, macos и т.п., то вам подойдет этот вариант. Если вы пользуетесь Windows, то для вас есть следующая секция.

Мы напишем shell скрипт, который будет делать то, что нам нужно. Для этого нужно создать файл с расширением `.sh`, в котором можно писать команды как в консоли, а также использовать циклы и т.п.

Для начала нужно скомпилировать все наши файлы (smart, stupid, gen и, возможно, checker). Рекомендуется при стресс-тестировании максимально приближаться к компиляции программ в тестирующей системе и использовать соответствующую оптимизацию (скорее всего, это `-O2`).

Далее мы запускаем цикл, в котором выполняем программы (компилируем мы один раз, а выполняем потом сколько угодно) и сравниваем результаты их работы.

Пример стресса:

```
g++ -std=c++17 -O2 smart.cpp -o smart
g++ -std=c++17 -O2 stupid.cpp -o stupid
g++ -std=c++17 -O2 gen.cpp -o gen

for t in $(seq 1 100000); do
    echo "Running test $t"
    ./gen $t > input
    ./smart < input > smart_output
    ./stupid < input > stupid_output
    diff smart_output stupid_output || exit
done
```

В первых трех строчках мы компилируем наши программы. Рассмотрим, к примеру, первую строчку. Здесь `g++` — это компилятор c++, `-std=c++17` устанавливает версию c++, под которой мы хотим компилировать программу, `-O2` устанавливает нужный уровень оптимизаций, а `-o smart` говорит о том, что мы хотим скомпилировать программу в файл под названием `smart` (обратите внимание, что в данном случае у него нет никакого расширения). Остальные две компиляции аналогичны.

Далее на 5 строке представлен код цикла, который перебирает переменную `t` от 1 до 100000. Обратите внимание, что цикл завершается словом `done` в конце скрипта. После чего при помощи команды `echo` мы выводим на экран номер теста, чтобы можно было отслеживать прогресс.

Затем запускаются программы. Здесь `./gen` означает запуск программы `gen`, `$t` означает, что мы передаем число `t` как параметр. Знак больше перенаправляет вывод программы. Таким образом, наш генератор будет писать тест не в консоль, а в файл под названием `input`. Затем мы аналогично запускаем два решения. Они пишут в файлы `smart_output` и `stupid_output`. При этом в этих строчках есть еще знак меньше, который наоборот означает то, откуда мы читаем входные данные. То, какой знак нужно использовать для чего, легко понять по направлению стрелочки. У знака `>` стрелочка указывает внутрь файла, то есть в этот файл мы выводим информацию, а у знака `<` стрелочка указывает наружу, то есть из этого файла мы достаем информацию.

После того, как программы отработали, настало время проверить совпадение ответов. Для этого мы воспользуемся консольной утилитой `diff`, которая сравнивает файлы на совпадение. Стоит обратить внимание, что если мы не пишем чекер и пользуемся `diff`'ом, то формат вывода у обеих программ должен совпадать с точностью до пробелов. Утилита `diff` возвращает 1, если файлы равны, и 0, если файлы не равны. Далее мы используем оператор `||`, который, как и в c++, означает «или». Если `diff` вернул 1, то результат «или» уже равен единице и второй аргумент не вычисляется. Если же было возвращено значение 0, то выполняется вторая команда `exit`, которая завершает наш скрипт. Кроме того, `diff` в этом случае выведет на экран места, в которых различаются файлы. Далее можно уже зайти в файл `input`, посмотреть, какой там тест, и пользоваться им для отладки программы. Если же неправильный тест не находится, то либо вы допустили ошибку при написании стресса, либо ваше решение верное, либо ошибка просто не ловится вашим стрессом (возможно, она проявляется только на больших тестах).

Чтобы запустить этот скрипт, нужно просто написать в консоль `./stress.sh`, если вы сохранили его в соответствующий файл. Однако, вероятно, вам выдадут строчку примерно такого содержания:

```
permition denied: ./stress.sh
```

Дело в том, что по умолчанию у файлов нет прав на исполнение. Так что нужно их добавить. Это можно сделать при помощи следующей команды:

```
chmod +x stress.sh
```

И затем попытаться запустить скрипт заново.

Теперь поговорим об улучшениях, которые можно сделать к нашему стрессу. Во-первых, может возникнуть ситуация, в которой какая-то из запускаемых программ упадет с ошибкой выполнения. Наш стресс это просто проигнорирует. Давайте это обработаем. Для этого просто добавим `|| exit` в конец запусков наших программ аналогично строке с `diff`:

```
1  g++ -std=c++17 -O2 smart.cpp -o smart
2  g++ -std=c++17 -O2 stupid.cpp -o stupid
3  g++ -std=c++17 -O2 gen.cpp -o gen
4
5  for t in $(seq 1 100000); do
6      echo "Running test $t"
7      ./gen $t > input || exit
8      ./smart < input > smart_output || exit
9      ./stupid < input > stupid_output || exit
10     diff smart_output stupid_output || exit
11 done
```

Теперь наша программа завершится, если какая-то из программ упала. Однако мы

не знаем, какая. Мы можем попытаться запустить их по очереди и проверить. Либо можно добавить описание причины ошибки:

```
g++ -std=c++17 -O2 smart.cpp -o smart
g++ -std=c++17 -O2 stupid.cpp -o stupid
g++ -std=c++17 -O2 gen.cpp -o gen

for t in $(seq 1 100000); do
    echo "Running test $t"
    ./gen $t > input || { echo "gen failed"; exit; }
    ./smart < input > smart_output || { echo "smart failed";
       exit; }
    ./stupid < input > stupid_output || { echo "stupid
       failed"; exit; }
    diff smart_output stupid_output || { echo "outputs aren't
       equal"; exit; }
done
```

29.3 Windows: Bat стрессы

Аналогичный пример с bat скриптами в Windows:

```
g++ -std=c++17 -O2 smart.cpp -o smart.exe
g++ -std=c++17 -O2 stupid.cpp -o stupid.exe
g++ -std=c++17 -O2 gen.cpp -o gen.exe

:beg
gen.exe > input || exit
smart.exe < input > smart_output
stupid.exe < input > stupid_output
fc smart_output stupid_output
if errorlevel 1 goto bug
goto beg
:bug
echo found!
```

Компиляция аналогична варианту с linux. Далее мы используем goto как циклы, бесконечно запуская тесты (здесь представлена вариация без входных параметров, то есть в коде должен использоваться часто меняющийся рандом). Далее используется команда `fc`, аналогичная `diff` в linux.

29.4 Python стрессы

В данном подходе мы используем вместо shell или bat скрипта более высокоуровневый вариант — питон. Из него мы можем запускать все те же консольные команды, но кроме того можем, к примеру, написать прямо там без труда свой собственный чекер или генератор (или не писать). Особых отличий от скриптовых вариантов больше нет, рекомендуется изучить код:

```
import os, sys

```

```
3  for i in range(1, 100000):
4      print('Running test', i)
5      os.popen('./gen > input')
6      smart_ans = os.popen('./smart < input').readlines()
7      stupid_ans = os.popen('./stupid < input').readlines()
8      if smart_ans != stupid_ans:
9          print('Outputs are not equal')
10         print('Input:')
11         print(*(open('input').readlines()))
12         print('stupid answer:')
13         print(*stupid_ans)
14         print('smart answer:')
15         print(*smart_ans)
16         sys.exit()
17 print('All tests passed')
```

Здесь `os.popen` выполняет консольную команду, а `readlines` сохраняет вывод программы.

Плюсом этого подхода является платформонезависимость.

29.5 In-Code стрессы

Наверное, самый простой вариант — сделать все в одной программе. Здесь smart, stupid, gen и checker будут не отдельными программами, а функциями. Вероятно также, что ввод-вывод в этом случае тестироваться не будут. Только смысловая часть программы. Если вы заранее не писали код в формате блоков: отдельная функция для ввода, отдельная функция для решения, отдельная функция для вывода, то вам придется немного поменять свое решение.

Однако плюсом данного подхода является его быстрота. Здесь мы не запускаем никаких программ, не пишем в файлы и т.д. В результате чего мы можем запускать миллионы тестов в секунду.

Если два решения используют функции или переменные с одинаковыми названиями, можно просто заключить оба решения в два разных пространства имен: `NamespaceSmart` и `NamespaceStupid`.

Пример такого стресса для задачи поиска квадрата числа можно посмотреть по ссылке.

30. Быстрый ввод-вывод в C++

Часто бывает так, что ввод и/или вывод в задаче очень большие. В этом случае только считывание входных данных и вывод ответа уже могут не уложиться в ограничение по времени. Чтобы этого избежать, есть различные техники, которые мы рассмотрим в этой главе.

30.1 endl и \n

Обычно перевод строки делается при помощи `cout << endl`, однако эта команда не только переводит строку, но и очищает буфер. Когда вы что-то выводите, на самом деле оно не обязано выводиться сразу. Программа сначала накопит какой-то буфер вывода, а потом выведет его целиком, потому что сама по себе операция вывода весьма долго работает. Однако если вы пишете `endl`, то вы заставляете программу насильно очистить буфер. Если вы будете делать это очень часто, то программа будет работать очень долго. Чтобы решить эту проблему, можно вместо `endl` использовать `cout << '\n'`. Отличие заключается как раз в том, что это просто символ перевода строки без очистки буфера. В случае, если вы в выводе часто делаете перевод строки, это может существенно ускорить вашу программу.

Однако не стоит везде всегда писать `\n` вместо `endl`. В интерактивных задачах важно, чтобы после каждого вывода очищался буфер, чтобы программа, с которой вы взаимодействовала, могла сразу считать то, что вы вывели. Поэтому в интерактивных задачах обязательно использовать `endl` для очистки буфера.

Кроме того, если вы пытаетесь найти ошибку в программе и выводите отладочную информацию, вам важно, чтобы она появлялась на экране, поэтому ее тоже нужно выводить с очисткой буфера. Если вы не хотите при этом переводить строку, можете воспользоваться `std::flush`.

30.2 sync_with_stdio и cin.tie

Также сильно ускорить программу можно, отключив синхронизацию разных потоков ввода/вывода. Можно использовать следующие две строчки:

```
1    ios::sync_with_stdio(0);
2    cin.tie(0);
```

Первая строка отключает синхронизацию `iostream` и `stdio`, то есть вы не можете больше одновременно использовать и `cin`, и `scanf` в одной программе. Однако вряд ли вам это нужно, поэтому эту синхронизацию можно отключить, что приводит к ускорению программы.

Вторая строка отключает привязку `cin` к `cout`, что помогает в случае, когда ввод и вывод чередуются. Кроме того, некоторые люди иногда используют `cout.tie(0)`, но эта строка не делает ничего, потому что пытается отключить привязку `cout` к `cout`, что весьма странно. Миф о том, что `cout.tie` полезен подкрепляется примерами, когда при добавлении этой строки код ускорялся, однако при повторных запусках могла наблюдаться обратная ситуация. Разное время работы связано с разными условиями запуска и фазами Венеры, но не с добавлением этой строчки.

30.3 Ручной быстрый ввод-вывод

Если даже предыдущие оптимизации не помогают, можно воспользоваться рукописным быстрым вводом. Это будет еще в несколько раз быстрее. К примеру, можно воспользоваться вот этим кодом от Сергея Копелиовича. Пример использования:

```
1    int n = readInt();
2    writeInt(n, ' ');
```

VIII

Дополнительные материалы