

Корневая оптимизация

Корневые эвристики — это обобщённое название различных методов и структур данных, опирающихся на тот факт, что если мы разделим какое-то множество из n элементов на блоки по \sqrt{n} элементов, то самих этих блоков будет не более \sqrt{n} .

Центральное равенство этой статьи: $\sqrt{x} = \frac{x}{\sqrt{x}}$.

Корневая декомпозиция на массивах

Сделаем вид, что про [дерево отрезков](#) мы не знаем, и рассмотрим следующую задачу:

Дан массив a длины n и q запросов одного из двух типов:

1. Найти сумму на отрезке $[l, r]$.
2. Увеличить все элементы на отрезке $[l, r]$ на x .

Разделим весь массив на блоки по $c \approx \sqrt{n}$ элементов и посчитаем сумму на каждом блоке. Так как блоки не пересекаются, суммарно это будет работать за $O(n)$.

```
// c это и количество блоков, и также их размер; оно должно быть чуть больше корня
const int maxn = 1e5, c = 330;
int a[maxn], b[c];
int add[c];

for (int i = 0; i < n; i++)
    b[i / c] += a[i];
```

Заведём также массив `add` размера \sqrt{n} , который будем использовать для отложенной операции прибавления на блоке. Будем считать, что реальное значение i -го элемента равно `a[i] + add[i / c]`.

Теперь мы можем отвечать на запросы первого типа за $O(\sqrt{n})$ на запрос:

1. Для всех блоков, лежащих целиком внутри запроса, просто возьмём уже посчитанные суммы и прибавим к ответу.
2. Для блоков, пересекающихся с запросом только частично (их два — правый и левый), проитерируемся по нужным элементам и прибавим к ответу.

```

int sum(int l, int r) {
    int res = 0;
    while (l <= r) {
        // если мы находимся в начале блока и он целиком в запросе
        if (l % c == 0 && l + c - 1 <= r) {
            res += b[l / c];
            l += c; // мы можем прыгнуть сразу на блок
        }
        else {
            res += a[l] + add[l / c];
            l += 1;
        }
    }
    return res;
}

```

Обновление пишется примерно так же:

```

void upd(int l, int r, int x) {
    while (l <= r) {
        if (l % c == 0 && l + c - 1 <= r) {
            b[l / c] += c * x;
            add[l / c] += x;
            l += c;
        }
        else {
            a[l] += x;
            l++;
        }
    }
}

```

Обе операции будут работать за $O(\sqrt{n})$, потому что нужных «центральных» блоков всегда не более \sqrt{n} , а в граничных блоках суммарно не более $2\sqrt{n}$ элементов.

Корневая эвристика на запросах

Сделаем вид, что запросов обновления нет. Тогда бы мы решали эту задачу просто массивом префиксных сумм.

```
int s[maxn]; // [0, r)
s[0] = 0

for (int i = 0; i < n; i++)
    s[i+1] = s[i] + a[i];
```

Подойдём теперь к задаче с другой стороны: разобьём на корневые блоки не массив, а запросы к нему. Будем обрабатывать каждый блок запросов независимо от других с помощью массива префиксных сумм, который мы будем честно пересчитывать каждые \sqrt{q} запросов.

На каждый запрос суммы мы можем потратить $O(1)$ времени на запрос к префиксным суммам, плюс \sqrt{q} времени на поправку на всех запросах из буфера.

```

struct query { int l, r, x; };
vector<query> buffer; // запросы, не учтенные в префиксных суммах

int sum(int l, int r) {
    int res = s[r + 1] - s[l];
    for (query q : buffer)
        // пересечем запрос суммы со всеми запросами
        res += q.x * max(0, min(r, q.r) - max(l, q.l));
    return res;
}

void rebuild() {
    vector<int> d(n, 0);
    // массив дельт

    for (query q : buffer) {
        d[q.l] += x;
        d[q.r + 1] -= x;
    }
    buffer.clear();

    int delta = 0, running_sum = 0;
    for (int i = 1; i <= n; i++) {
        p[i] += running_sum;
        delta += d[i];
        running_sum += delta;
    }
}

void upd(int l, int r, int x) {
    buffer.push_back({l, r, x});
    if (buffer.size() == c)
        // буфер слишком большой; надо пересчитать префиксные суммы и очистить его
        rebuild();
}

```

Такое решение будет работать за $O(n\sqrt{q} + q\sqrt{q})$.

Преобразование статических структур в динамические

Эту технику можно применять не только к массиву префиксных сумм, но и к любым статическим структурам данных.

Требуется добавлять точки в выпуклую оболочку и уметь находить касательные (то есть находить точку, которая максимизирует скалярное произведение $a_i x + b_i y$).

Мы можем так же каждые \sqrt{q} запросов перестраивать выпуклую оболочку, а при ответе на запрос касательной помимо кандидата из построенной выпуклой оболочки рассмотреть дополнительно \sqrt{q} скалярных произведений из точек из буфера.

Теперь чуть сложнее:

Требуется добавлять и удалять точки из выпуклой оболочки и уметь находить касательные. Запросы известны заранее.

Разобьём запросы на блоки явно, и будем обрабатывать их по отдельности. На каждом блоке построим выпуклую оболочку только тех точек, которые существуют на всём её блоке, а остальные запросы сохраним в буфер. При ответе на запрос касательной помимо кандидата из построенной оболочки будем рассматривать все точки, которые существуют на момент данного запроса — найти все такие точки можно за \sqrt{q} , проанализировав историю текущего блока.

Часто, если удалений в задаче нет или их можно как-то эмулировать, можно применить похожую, но более мощную технику: поддерживать $O(\log n)$ структур (например, выпуклых оболочек) размеров степени двойки, причём так, что нет двух структур одинакового размера. При добавлении новой точки мы создаём для неё новую структуру размера 1. Далее, если структура размера 1 уже существует, то мы объединяем эти две структуры и создаём структуру размера 2. Если структура размера 2 уже существовала, то мы объединяем её и создаём структуру размера 4, и так далее. Запрос мы будем передавать во все $O(\log n)$ базовых структур и объединять ответы.

«Алгоритм Мо»

Обсудим ещё одно решение этой же задачи: на этот раз будем группировать запросы и по временным, и по пространственным признакам. Пусть все запросы даны нам заранее, и запросов обновления нет. Опять же притворимся, что про префиксные суммы мы не слышали.

Сгруппируем все запросы в корневые блоки по их левой границе, внутри каждого блока отсортируем запросы по правой границе и будем обрабатывать каждый такой блок по отдельности. Будем проходиться по запросам блока, поддерживая сумму текущего отрезка. Для первого отрезка посчитаем сумму честно, а для всех остальных будем каждый раз сдвигать его границы и обновлять сумму — по одному элементу за раз.

Трюк в том, что правая граница суммарно сдвинется на $O(n)$, потому что отрезки отсортированы, а левая — каждый раз на $O(\sqrt{n})$. Итоговая асимптотика решения будет $O(q\sqrt{n} + n\sqrt{n})$: изменение левых границ суммарно по всем блокам займёт $O(q\sqrt{n})$ операций, а правых — $O(n\sqrt{n})$,

```

struct query { int l, r, idx; };
// idx -- это номер запроса, на который нужно ответить

int a[maxn];
vector<query> b[c];
int ans[maxn]; // массив ответов на запросы

// где-то в main:

for (query q : queries)
    b[q.l / c].push_back(q);

for (int i = 0; i < c; i++) {
    sort(b[i].begin(), b[i].end(), [](query a, query b){
        return a.r < b.r;
    });
}

for (int i = 0; i < c; i++) {
    int l = i * c, r = i * c - 1; // границы текущего отрезка
    int s = 0; // сумма на текущем отрезке
    for (query q : b[i]) {
        while (r < q.r)
            s += a[++r];
        while (l < q.l)
            s -= a[l++];
        while (l > q.l)
            s += a[--l];
        ans[q.idx] = s;
    }
}

```

Конкретно для этой задачи этот алгоритм не нужен, однако он полезен для ответа на более сложные вопросы: найти k -ую порядковую статистику на отрезке, его медиану, количество различных элементов, наиболее часто встречающийся элемент и так далее. Во всех этих случаях нужно просто вместо суммы поддерживать какую-нибудь структуру — например, хэш-таблицу — отвечающую за множество элементов на отрезке, и пересоздавать её между блоками.

Примечание. Алгоритм назван по имени какого-то неизвестного китайца и принят под этим названием в фольклоре ICPC. Автору это не нравится, но что поделать: другого названия не придумали.

Вариации

На деревьях. В некоторых задачах требуется считать результаты операций на дереве. Например: рядом с каждой вершиной есть число, и нам нужно возвращать количества различных значений на путях.

Идея примерно такая же, как при [сведении LCA к RMQ](#). Мы можем выписать эйлеров обход дерева (каждую вершину выписываем дважды — в моменты входа и выхода), и теперь задача сводится к задаче на массиве, только с одним исключением: мы должны добавлять в структуру только те элементы, которые содержатся на отрезке ровно 1 раз.

Использовать корневую декомпозицию как структуру. Часто от внутренней структуры в алгоритме требуется что-то сложное, и мы прибегаем к использованию каких-то «тяжелых» структур, вроде дерева отрезков. Но данный случай отличается от обычных задач на обработку запросов: у нас обновлений $O(n\sqrt{n})$, а запросов всего $O(n)$. Здесь эффективнее вместо дерева отрезков, требующего $O(\log n)$ времени на оба типа запросов, взять структуру, которая быстро работает при обновлениях, и не очень быстро на самих запросах — например, корневую декомпозицию, которая работает за $O(1)$ и $O(\sqrt{n})$ соответственно.

На самом деле, это примерно единственное место, где корневую эвристику как структуру использовать эффективнее всего — внутри другой корневой эвристики.

«3D Мо». Если очень захотеть, этот подход можно применять и в задачах, где надо обрабатывать запросы обновления. Для этого нужно ввести третий параметр get-запроса t , который будет равен числу update-запросов до текущего get.


Снова отсортируем все get-запросы, но на этот раз в порядке $(\frac{t}{n^{\frac{2}{3}}}, \frac{l}{n^{\frac{2}{3}}}, r)$, и обработаем их таким же алгоритмом, только в трёх измерениях. Время работы нового подхода будет $O(n^{\frac{5}{3}})$, что доказывается аналогично исходному алгоритму.

Деление на тяжелые и легкие объекты

Всем известный алгоритм факторизации за корень опирается на тот факт, что каждому «большому» делителю $d \geq \sqrt{n}$ числа n соответствует какой-то «маленький» делитель $\frac{n}{d} \leq \sqrt{n}$.

Подобное полезное свойство (что маленькие объекты маленькие, а больших объектов не много) можно найти и у других объектов.

Длинные и короткие строки

 [710F. Операции над множеством строк](#). Требуется в онлайн обрабатывать три типа операций над множеством строк:

1. Добавить строку в множество.
2. Удалить строку из множества.
3. Для заданной строки, найти количество её вхождений как подстроки среди всех строк множества.

Одно из решений следующее: разделим все строки на *короткие* ($|s| < \sqrt{l}$) и *длинные* ($|s| \geq \sqrt{l}$), где l означает суммарную длину всех строк. Заметим, что длинных строк немного — не более \sqrt{l} .

С запросами будем справляться так:

- Заведём хэш-таблицу, и когда будем обрабатывать запрос добавления или удаления, будем прибавлять или отнимать соответственно единицу по [хэшам](#) всех её коротких подстрок. Это можно сделать суммарно за $O(l\sqrt{l})$: для каждой строки нужно перебрать $O(\sqrt{l})$ разных длин и окном пройти по всей строке.
- Для запроса третьего типа для короткой строки, просто посчитаем её хэш и посмотрим на значение в хэш-таблице.
- Для запроса третьего типа для длинной строки, мы можем позволить себе посмотреть на все неудалённые строки, потому что таких случаев будет немного, и если мы можем за линейное время найти все вхождения новой строки, то работать это будет тоже за $O(l\sqrt{l})$. Например, можно посчитать [z-функцию](#) для всех строк вида $s\#t$, где s это строка из запроса, а t это строка из множества; здесь, правда, есть нюанс: s может быть большой, а маленьких строк t много — нужно посчитать z-функцию сначала только от s , а затем виртуально дописывать к ней каждую t и досчитывать функцию.

Иногда отдельный подход к тяжелым и лёгким объектам не требуется, но сама идея помогает увидеть, что некоторые простые решения работают быстрее, чем кажется.

Треугольники в графе

Рассмотрим другую задачу:

Дан граф из n вершин и $m \approx n$ рёбер. Требуется найти в нём количество циклов длины три.

Будем называть вершину *тяжелой*, если она соединена с более чем \sqrt{n} другими вершинами, и *лёгкой* в противном случае.

Попытаемся оценить количество соединённых вместе троек вершин, рассмотрев все возможные 4 варианта:

0. В цикле нет тяжелых вершин. Рассмотрим какое-нибудь ребро (a, b) цикла. Третья вершина c должна лежать в объединении списков смежности g_a и g_b , а раз обе эти вершины лёгкие, то таких вершин найдётся не более \sqrt{n} . Значит, всего циклов этого типа может быть не более $O(m\sqrt{n})$.
1. В цикле одна тяжелая вершина. Аналогично — есть одно «лёгкое» ребро, а значит таких циклов тоже $O(m\sqrt{n})$.
2. В цикле две тяжелые вершины — обозначим их как a и b , а лёгкую как c . Зафиксируем пару (a, c) — способов это сделать $O(m)$, потому что всего столько рёбер. Для этого ребра будет не более $O(\sqrt{n})$ рёбер (a, b) , потому что столько всего тяжелых вершин. Получается, что всего таких циклов может быть не более $O(m\sqrt{n})$.
3. Все вершины тяжелые. Аналогично — тип третьей вершины в разборе предыдущего случая нигде не использовался; важно лишь то, что тяжелых вершин b немного.

Получается, что циклов длины 3 в графе может быть не так уж и много — не более $O(m\sqrt{n})$.

Само решение максимально простое: отсортируем вершины графа по их степени, ориентируем ребра $v \rightarrow u, v \leq u$; теперь внутренним циклом будем перебирать пути $v \rightarrow u \rightarrow w, v \leq u \leq w$, а потом проверять существование ребра $v \rightarrow w$.

```

vector<int> g[maxn], p(n); // исходный граф и список номеров вершин
iota(p.begin(), p.end(), 0); // 0, 1, 2, 3, ...

// чтобы не копиастить сравнение:
auto cmp = [&](int a, int b) {
    return g[a].size() < g[b].size() || (g[a].size() == g[b].size() && a < b);
};

// в таком порядке мы будем перебирать вершины
sort(p.begin(), p.end(), cmp);

// теперь удалим все лишние рёбра (ведущие в более тяжелые вершины)
for (int v = 0; v < n; v++) {
    vector<int> &t = g[v];
    // отсортируем их и удалим какой-то суффикс
    sort(t.begin(), t.end(), cmp);
    while (t.size() > 0 && cmp(t.back(), v))
        t.pop_back();
    reverse(t.begin(), t.end());
}

// рядом с каждой вершиной будем хранить количество
// ранее просмотренных входящих рёбер (v -> w)
vector<int> cnt(n, 0);
int ans = 0;

for (int v : p) {
    for (int w : g[v])
        cnt[w]++;
    for (int u : g[v])
        for (int w : g[u])
            ans += cnt[w]; // если в графе нет петель, то cnt[w] это 0 или 1
    // при переходе к следующему v массив нужно занулить обратно
    for (int w : g[v])
        cnt[w]--;
}

```

Задачу также можно было решить чуть более прямолинейно, переместив все проверки внутрь основного цикла, но это сказало бы на скорости работы.

Рюкзак за $O(S\sqrt{S})$

Если у нас есть n предметов с весами w_1, w_2, \dots, w_n , такими что $\sum w_i = S$, то мы можем решить задачу о рюкзаке за время $O(S \cdot n)$ стандартной динамикой. Чтобы решить задачу быстрее, попытаемся сделать так, чтобы число предметов n стало $O(\sqrt{S})$.

Заметим, что количество различных весов среди будет $O(\sqrt{S})$, потому что если среди них есть k различных чисел, то:

$$S = w_1 + w_2 + \dots + w_n \geq 1 + 2 + \dots + k = \frac{k \cdot (k + 1)}{2}$$

Откуда значит, что $k \leq 2\sqrt{S}$.

Рассмотрим теперь некоторый вес x , который k раз встречается в наборе весов. «Разложим» k по степеням двойки и вместо всех k вхождений этого веса добавим веса $x, 2 \cdot x, 4 \cdot x, \dots, (c - 1 - 2^t) \cdot x$, где t это максимальное целое число, для которого выполняется $2^t - 1 \leq c$. Легко видеть, что все суммы вида $q \cdot x$ ($q \leq k$) и только их по-прежнему можно набрать.

Алгоритм в этом и заключается — проведем данную операцию со всеми уникальными значениями весов и после чего запустим стандартное решение. Уже сейчас легко видеть, что новое количество предметов будет $O(\sqrt{S} \log S)$, потому что для каждого веса мы оставили не более $\log S$ весов, а всего различных весов было $O(\sqrt{S})$.

Упражнение. Доказать, что предметов на самом деле будет $O(\sqrt{S})$.

Примечание. Классическое решение рюкзака можно ускорить на несколько порядков, если использовать [bitset](#) или [векторизовать](#) его основной цикл.

Бакеты

Пусть нам нужно находить всё ту же сумму, но теперь у нас запрос обновления выглядит так: i -тый элемент *прыгает* на d позиций вправо или влево, смещая, соответственно, d элементов между начальной и конечной позицией на единицу вправо или влево. Сумму нужно находить уже в новой индексации. Про [декартово дерево](#) автор снова просит читателя на время забыть.

Объединим предыдущие подходы: теперь не только будем делить массив на блоки *примерно* по корню, но и каждый корень запросов будем перестраивать всю структуру.

А именно, теперь мы будем работать не с чётко разделённые блоки, а с *корзинами* переменного размера, в которых хранятся небольшие массивы с элементами, а также сумма на всей корзине. До начала обработки запросов добавим в каждую из них примерно по \sqrt{n} последовательных элементов.

При нахождении суммы мы будем действовать как раньше, а при запросе обновления найдём две нужные корзины (из какой корзины элемент удаляется и в какую добавляется) и полностью их пересоздадим — за их суммарный размер. Чтобы корзины не стали слишком большими, просто будем каждые \sqrt{n} запросов их полностью перестраивать.

Такой подход всё ещё будет работать за $O(m\sqrt{n})$, но позволяет решать задачи, в которых меняется порядок элементов. Реальный пример: IOI 2011 «[Dancing Elephants](#)».

Подбор константы

На скорость работы очень сильно влияет размер блока. Мы для простоты использовали одну и ту же константу и для количества блоков, и для их размера, но на практике их часто нужно подбирать.

Иногда асимптотики «тяжелой» и «лёгкой» части получаются разными, потому что мы где-то не смогли обойтись без какой-нибудь структуры, которая внесла лишний логарифм. Чаще всего, в качестве оптимального размера блока c^* можно взять что-то близкое к решению уравнения $g \cdot \frac{n}{c} = f \cdot c$, где f и g это асимптотики блочной и не-блочной частей соответственно.

Впрочем, надо также учитывать, что походы в какое-нибудь декартово дерево совсем не в логарифм раз медленнее линейного прохода по массиву.