Пересчёт динамики по слоям

В этой задаче мы рассмотрим 4 связанных между собой способа оптимизации динамики. Во всех четырёх мы будем решать одну и ту же задачу:

> Даны n точек на прямой. Нужно найти m отрезков, покрывающих все точки, минимизировав при этом сумму квадратов их длин.

Базовое решение — это следующая динамика:

- $f[i, j]$ — минимальная стоимость покрытия i первых (самых левых) точек, используя не более j отрезков.
- Переход — перебор всех возможных последних отрезков, то есть $f[i, j] = \min_{k<i}\{f[k, j-1] + (x_{i-1} - x_k)^2\}$.

Итоговый ответ будет записан в $f[n, m]$, а такое решение непосредственным перебором будет работать за $O(n^2 m)$.

```
// x[] - отсортированный массив координат точек, индексация с нуля

// квадрат длины отрезка с i-той до j-той точки
int cost(int i, int j) { return (x[j]-x[i])*(x[j]-x[i]); }

for (int i = 0; i <= m; i++)
    f[0][k] = 0; // если нам не нужно ничего покрывать, то всё и так хорошо
// все остальные f предполагаем равными бесконечности

for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++)
        for (int k = 0; k < i; k++)
            f[i][j] = min(f[i][j], f[k][j-1] + cost(k, i-1));
```

Заметим, что циклы по `i` и `j` можно поменять местами.

Разделяй-и-властвуй

Обозначим за $opt[i, j]$ оптимальный k, то есть тот, на котором $f[i, j] = f[k, j-1] + (x_{i-1} - x_k)^2$ минимизируется. Для однозначности, если оптимальный индекс не один, то выберем среди них самый правый.

Утверждение. $opt[i, j] \leq opt[i, j+1]$.

Интуиция такая: если у нас появился дополнительный отрезок, то последний отрезок нам не выгодно делать больше.

Что это нам даёт? Если мы уже знаем $opt[i,l]$ и $opt[i,r]$ и хотим посчитать $opt[i,j]$ для какого-то j между l и r, то мы можем сузить отрезок поиска оптимального индекса со всего $[0,i-1]$ до $[opt[i,l], opt[i,r]]$.

Будем делать следующее: заведем рекурсивную функцию, которая считает динамики для отрезка $[l,r]$, зная, что их opt-ы лежат между l' и r'. Она берет середину отрезка $[l,r]$ и линейным проходом считает ответ для неё, а затем просто спускается дальше рекурсивно.

```
void solve(int l, int r, int _l, int _r, int k) {
    if (l > r)
        return; // отрезок пустой -- выходим
    int t = (l + r) / 2, opt = _l;
    for (int i = _l; i <= min(_r, t); i++) {
        int val = f[i+1][k-1] + cost(i, j);
        if (val < f[t][k])
            f[t][k] = val, opt = i;
    }
    solve(l, t-1, _l, opt, k);
    solve(t+1, r, opt, _r, k);
}
```

Затем последовательно вызовем её для каждого слоя:

```
for (int k = 1; k <= m; k++)
    solve(0, n-1, 0, n-1, k);
```

Асимптотика. Теперь пересчет одного «слоя» динамики занимает $O(n \log n)$ вместо $O(n^2)$, потому что каждый раз рекурсивная функция уменьшает в два раза хотя бы один из отрезков. Так как максимальная глубина рекурсии будет $O(\log n)$, то каждый элемент будет просмотрен не более $O(\log n)$ раз.

Таким образом, мы улучшили асимптотику до $O(nm \log n)$.

Оптимизация Кнута

Предыдущий метод опирался на тот факт, что $opt[i,j] \leq opt[i,j+1]$. Асимптотику можно ещё улучшить, если opt монотонен ещё и по первому параметру:

$$opt[i-1,j] \leq opt[i,j] \leq opt[i,j+1]$$

В задаче это выполняется примерно по той же причине: если нам нужно покрывать меньше точек, то последний отрезок будет начинаться не позже старого.

Будем просто для каждого состояния перебирать элементы непосредственно от $opt[i-1,j]$ до $opt[i,j+1]$ — можно идти в порядке увеличения i и уменьшения j, и тогда эти opt уже будут посчитаны к нужному моменту.

Выясняется, что это работает быстро. Чтобы понять, почему, распишем количество элементов, которые мы просмотрим для каждого состояния, и просуммируем:

$$\sum_{i,j}(opt[i,j+1]-opt[i-1,j]+1)=nm+\sum_{ij}(opt[i,j+1]-opt[i-1,j])$$

Заметим, что все элементы, кроме граничных, учитываются в сумме ровно два раза — один раз с плюсом, другой с минусом — а значит их можно сократить. Граничных же элементов $O(n)$ и каждый из них порядка $O(n)$. Значит, итоговая асимптотика составит $O(n\cdot m+n\cdot n)=O(n^2)$.

```
for (int i = 1; i <= n; i++) {
    for (int j = m; j >= 1; j--) {
        for (int k = opt[i-1][j]; k <= opt[i][j+1]; k++) {
            int val = f[i+1][k-1] + cost(i, j);
            if (val < f[t][k])
                f[t][k] = val, opt[i][j] = i;
        }
    }
}
```

Реализация получилась очень лаконичной: она всего на 3 строчки длиннее, чем базовое решение.

Convex Hull Trick

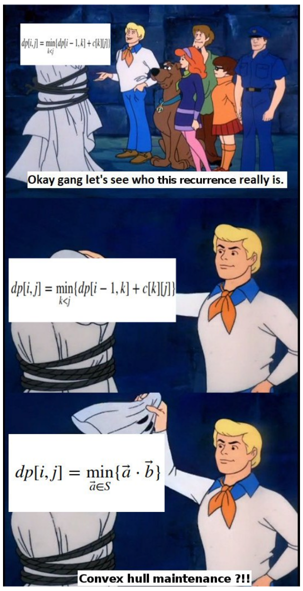
Возьмём исходную формулу для f и раскроем скобки в `cost`:

$$f[i,j]=\min_{k<i}\{f[k,j-1]+(x_{i-1}-x_k)^2\}=\min_{k<i}\{f[k,j-1]+x_{i-1}^2-2x_{i-1}x_k+x_k^2\}$$

Заметим, что x_{i-1}^2 не зависит от k, значит его можно вынести. Под минимумом тогда останется

$$\underbrace{f[k,j-1]+x_k^2}_{a_k}\underbrace{-2x_k}_{b_k}x_{i-1}$$

Это выражение можно переписать как $\min_k(a_k,b_k)\cdot(1,x_{i-1})$, где под «·» имеется в виду скалярное произведение.



Пусть мы хотим найти оптимальное k для $f[i][j]$. Представим все уже посчитанные релевантные динамики с предыдущего слоя как точки (a_k, b_k) на плоскости. Чтобы эффективно находить среди них точку с минимальным скалярным произведением, можно поддерживать их *нижнюю огибающую* (вектор $(1, x_{i-1})$ «смотрит» всегда вверх, поэтому нам интересна только она) и просто бинпоиском будем находить оптимальную точку.

Хранить нижнюю огибающую можно просто в стеке. Так как добавляемые точки отсортированы по x, её построение будет занимать линейное время, а асимптотика всего алгоритму будет упираться в асимптотику бинарного поиска, то есть будет равна $O(nm \log n)$

```
struct line {
    int k, b;
    line() {}
    line(int a, int _b) { k = a, b = _b; }
    int get(int x) { return k * x + b; }
};

vector<line> lines; // храним прямые нижней огибающей
vector<int> dots; // храним x-координаты точек нижней огибающей
//      ^ первое правило вещественных чисел
//      считаем, что в dots лежит округленная вниз x-координата

int cross(line a, line b) { // считаем точку пересечения
                            // считаем a.k > b.k
    int x = (b.b - a.b) / (a.k - b.k);
    if (b.b < a.b) x--; // боремся с округлением у отрицательных чисел
    return x;
}


void add(line cur) {
    while (lines.size() && lines.back().get(dots.back()) > cur.get(dots.back())) {

        lines.pop_back();
        dots.pop_back();
    }
    if (lines.empty())
        dots.push_back(-inf);
    else
        dots.push_back(cross(lines.back(), cur));
    lines.push_back(cur);
}

int get(int x) {
    int pos = lower_bound(dots.begin(), dots.end(), x) - dots.begin() - 1;
    return lines[pos].get(x);
}
```

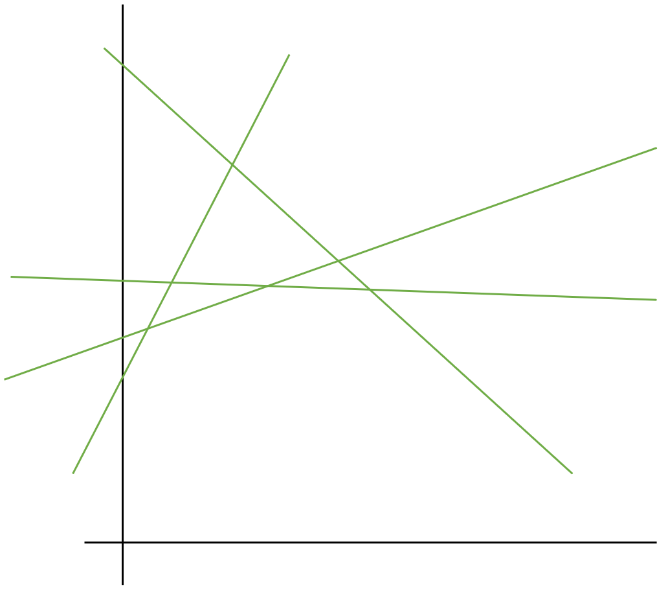
В случае нашей конкретной задачи, алгоритм можно и дальше соптимизировать, если вспомнить, что $opt[i, j] \leq opt[i][j+1]$, то есть что оптимальная точка всегда будет «правее». Это позволяет вместо бинпоиска применить метод двух указателей:

```
// TODO: закомитьте кто-нибудь реализацию на гитхаб
```

Мы избавились от бинпоиска, и теперь алгоритм работает за $O(n \cdot m)$.

Дерево Ли Шао

Существует другой подход к Convex Hull Trick: увидеть здесь не точки и оптимизацию скалярного произведения, а линии и нахождение минимума в точке среди этих линий.



Применительно к нашей задаче, выражение $\min_k(a_k, b_k) \cdot (1, x_{i-1})$ можно раскрыть как $\min_k(a_k + b_k \cdot x_{i-1})$ и представить как нахождение минимума в точке среди множества прямых вида $y = a_k + b_k \cdot x$.

Дерево Ли Шао (англ. *Li Chao segment tree*, кит. 李超段树) — модификация дерева отрезков над множеством возможных x, каждая вершина которого хранит в себе такую прямую, что если пройти по пути от корня до соответствующего листа, то максимум в данной точке будет наибольшее значение на пути.

Пусть в вершину пришло обновление — прямая *new*. Если в ней ничего не хранится, то запишем *new* в вершину и выйдем. Если там уже есть какая-то другая прямая *old*, то одна из них будет «доминировать» над другой хотя бы на одной из половин, а в другой будет либо пересекаться, либо тоже доминировать.

Если одна прямая полностью доминирует над другой, то мы её просто запишем в вершину, а про вторую забудем. Если же прямая доминирует только в одной из половин, то мы запишем её, а «проигравшую» прямую передадим в рекурсию в ту половину, где она может доминировать.

```
typedef int ftype;
typedef complex<ftype> point;
#define x real
#define y imag

ftype dot(point a, point b) {
    return (conj(a) * b).x();
}

ftype f(point f, ftype x) {
    return dot(f, {x, 1});
}

const int inf = 1e6 + 42;

point ln[8 * inf];
void add_line(point nw, int v = 1, int l = -inf, int r = inf) {
    point ol = ln[v];
    int m = (l + r) / 2;
    bool lef = f(nw, l) > f(ol, l);
    bool mid = f(nw, m) > f(ol, m);
    ln[v] = mid ? nw : ol;
    if(r - l == 1)
        return;
    if(lef != mid)
        add_line(mid ? ol : nw, 2 * v, l, m);
    else
        add_line(mid ? ol : nw, 2 * v + 1, m, r);
}

int get(int x, int v = 1, int l = -inf, int r = inf) {
    if(r - l == 1)
        return f(ln[v], x);
    int m = (l + r) / 2;
    if(x < m)
        return max(f(ln[v], x), get(x, 2 * v, l, m));
    else
        return max(f(ln[v], x), get(x, 2 * v + 1, m, r));
}
```

Достаточно полезно сравнить между собой CHT и дерево Ли-Шао и понимать, в какой из ситуаций стоит применять каждую из этих структур. Адекватные реализации CHT требуют особых условий — точки должны быть отсортированы по x. Если это выполнено, то работать алгоритм будет значительно быстрее, чем дерево Ли Шао, которое, в свою очередь, решает более общую задачу, но работает за $O(\log MAXC)$ на запрос, а зачастую еще и требует неявную реализацию, если $MAXC$ достаточно большое (неявная реализация схожа с неявным деревом отрезков).

Лямбда-оптимизация

Примечание. В научной литературе метод известен как дискретный метод множителей Лагранжа.

Рассмотрим немного другую задачу. Пусть нам нужно покрыть те же точки, но теперь нас не ограничивают жёстко в количестве отрезков, а просто штрафуют на какую-то константу λ за использование каждого. Нашу оптимизируемую функцию g можно выразить через f следующим образом:

$$g[i] = \min_{k<i}\{f[i,k] + k \cdot \lambda\}$$

Однако её можно считать по более оптимальной формуле, не сводя к вычислению f:

$$g[i] = \lambda + \min_{k<i}\{g[k] + (x_{i-1} - x_k)^2\}$$

Эту динамику можно посчитать за $O(n)$ — мы это делали полстраницы назад с помощью Convex Hull Trick.

Наблюдение 1. Если в оптимальном решении для g_i мы для какого-то λ использовали ровно k отрезков, то это же решение будет оптимальным и для $f[i][k]$.

Наблюдение 2. Если уменьшать λ, то оптимальное количество отрезков для для g_i будет увеличиваться.

Основная идея оптимизации: сделаем бинпоиск по λ, внутри которого будем находить оптимальное решение для g_i с таким λ. Если оптимальное k больше j, то следующая λ должна быть меньше, а в противном случае наоборот. Когда k совпадёт с j, просто выведем «чистую» стоимость получившегося решения.

Таким образом, задача решается за $O(n \log n + n \log m)$, если сортировку точек для CHT делать заранее, а не внутри бинпоиска.

Мы не учли только одну деталь: почему вообще существует такая λ, что оптимальное $k = j$. Возможно, что функция $k(\lambda)$ через него «перескакивает». В общем случае это действительно проблема: одной лишь монотонности не достаточно, чтобы решать подобным образом произвольные задачи с ограничением на число объектов.

Утверждение. Функция $f[i,j]$ *нестрого вогнутая* (то есть выпуклая вверх) по своему второму аргументу, то есть:

$$f[i,j] - f[i,j-1] \leq f[i,j+1] - f[i,j]$$

Иными словами, «выгода» добавления следующего отрезка с каждым разом не увеличивается. Тогда если мы найдем минимальную λ такую, что $k \geq j$, то $f[i,k] = f[i,j]$.

```
pair<ll, int> dp[maxn]; // dp[i] - (ответ, число отрезков)

void init() {
    for (int i = 0; i < maxn; i++) {
        dp[i] = make_pair(inf, 0);
    }
}

pair<ll, int> check(ll x) { // это можно соптимизировать
    init();
    dp[0] = make_pair(0ll, 0); // 1-индексация
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j < i; j++) {
            dp[i] = min(dp[i], {dp[j].first + cost[j + 1][i] + x, dp[j].second + 1});

        }
    }
    return dp[n];
}

ll solve() {
    ll l = -1e14; // границы надо подбирать очень аккуратно!
    ll r = 1;
    while (l + 1 < r) {
        ll mid = (l + r) / 2;
        pair<ll, int> x = check(mid);
        if (x.second >= k) {
            l = mid;
        }
        else {
            r = mid;
        }
    }
    pair<ll, int> result = check(l);
    return result.first - l * return.second; // вычитаем штрафы
}
}
```

Суммируем

TODO: сделать табличку

- Разделяйка: $O(nm \log n)$, если `cost` такой, что `opt` монотонна по одному аргументу.
- Кнут: $O(nm)$, если `cost` такой, что `opt` монотонна по обоим аргументам.
- CHT: $O(nm)$. В оптимизируемой функции нужно увидеть скалярное произведение.
- Лагранж: $O(n \log n)$. Функция должна быть выпуклой.