1 Vim config

```
filetype on
filetype plugin on
filetype plugin indent on

colorscheme ron

set nobackup
set noswapfile
set showcmd
set incsearch
set smartcase
set hidden
set lazyredraw
set nocompatible
set backspace=indent,eol,start
set history=10
set ruler
set expandtab
set shiftwidth=4
set softtabstop=4
set tabstop=4
set foldmethod=syntax
set virtualedit=all
set formatoptions=tcqrn
set wildmenu
set shm=aoOAI
set hlsearch
set number
set mousehide
set mouse=a
set termencoding=utf-8
set novisualbell
set encoding=utf-8
set fileencodings=utf8,cp1251

command W w
command Q q
command Wq wq
command WQ wq
command Wa wa
command WA wa
command Wqa wqa
command WQa wqa
command WQA wqa

# Mappings
map Q gq
map Y y$
# Save all files
map <F2> <Esc>:wa<CR>
# Save all files and exit
map <F3> <Esc>:wqa<CR>
# Replace template of current word
map <F4> :%s/\<<C-r>=expand("<cword>")<cr>\>/

nmap ,h :tabprev<CR>
nmap ,l :tabnext<CR>
nmap ,j :bnext<CR>
nmap ,k :bprevious<CR>
nmap ,p :set paste!<CR>
nmap ,t :tabnew<CR>
nmap <S-Tab> <<
nmap <Tab> <C-W>w
nmap <Backspace> <Esc>hx<Ins>
nmap <CR> i<CR><Esc>l

imap <S-Tab> <Esc><<i
imap <Ins> <Esc>a

vmap <Tab> >gv
vmap <S-Tab> <gv
```

2 Inverse of modulo ring

```
int *im;

// Initialization
im = (int *)calloc(N+1, sizeof(int));
for (int i = 0; i < N; ++i) {
    im[i] = -1;
```

```
}

int64_t inverse(int64_t a, int64_t n) {
    if (im[a] > 0)
        return im[a];

    int64_t tmp;
    int64_t t = 0, t1 = 1;
    int64_t r = n, r1 = a;
    while (r1 != 0) {
        int64_t q = r / r1;
        tmp = t - q * t1; t = t1; t1 = tmp;
        tmp = r - q * r1; r = r1; r1 = tmp;
    }
    if (t < 0)
        t += n;
    t = (t + n) % n;
    im[a] = t;
    return t;
}
```

3 Fenwick tree

```
vector<int> t;
int n;

int sum (int r)
{
    int result = 0;
    for (; r >= 0; r = (r & (r+1)) - 1)
        result += t[r];
    return result;
}

void inc (int i, int delta)
{
    for (; i < n; i = (i | (i+1)))
        t[i] += delta;
}
```

4 RMQ

Простая РМКу с обновлением на отрезке.

```
void build (int a[], int v, int tl, int tr) {
    if (tl == tr)
        t[v] = a[tl];
    else {
        int tm = (tl + tr) / 2;
        build (a, v*2, tl, tm);
        build (a, v*2+1, tm+1, tr);
    }
}

void update (int v, int tl, int tr, int l, int r, int add) {
    if (l > r)
        return;
    if (l == tl && tr == r)
        t[v] += add;
    else {
        int tm = (tl + tr) / 2;
        update (v*2, tl, tm, l, min(r,tm), add);
        update (v*2+1, tm+1, tr, max(l,tm+1), r, add);
    }
}

int get (int v, int tl, int tr, int pos) {
    if (tl == tr)
        return t[v];
    int tm = (tl + tr) / 2;
    if (pos <= tm)
        return t[v] + get (v*2, tl, tm, pos);
    else
        return t[v] + get (v*2+1, tm+1, tr, pos);
}
```

5 Treap

Ключевая идея заключается в том, что в качестве ключей key следует использовать индексы элементов в массиве. Однако явно хранить эти значения key мы не будем (иначе, например, при вставке элемента пришлось бы изменять key в O (N) вершинах дерева).

Заметим, что фактически в данном случае ключ для какой-то вершины - это количество вершин, меньших неё. Следует заметить, что вершины, меньшие данной, находятся не только в её левом поддереве, но и, возможно, в левых поддеревьях её предков. Более строго, неявный ключ для некоторой вершины t равен количеству вершин cnt(t->l) в левом поддереве этой вершины плюс аналогичные величины cnt(p->l)+1 для каждого предка p этой вершины, при условии, что t находится в правом поддереве для p.

Ясно, как теперь быстро вычислять для текущей вершины её неявный ключ. Поскольку во всех операциях мы приходим в какую-либо вершину, спускаясь по дереву, мы можем просто накапливать эту сумму, передавая её функции. Если мы идём в левое поддерево - накапливаемая сумма не меняется, а если идём в правое - увеличивается на cnt(t->l)+1. Теперь перейдём к реализации различных дополнительных операций на неявных декартовых деревьях:

- Вставка элемента. Пусть нам надо вставить элемент в позицию pos. Разобьём декартово дерево на две половинки: соответствующую массиву [0..pos-1] и массиву [pos..sz]; для этого достаточно вызвать split (t, t1, t2, pos). После этого мы можем объединить дерево t1 с новой вершиной; для этого достаточно вызвать merge (t1, t1, new_item) (нетрудно убедиться в том, что все предусловия для merge выполнены). Наконец, объединим два дерева t1 и t2 обратно в дерево t - вызовом merge (t, t1, t2).

- Удаление элемента. Здесь всё ещё проще: достаточно найти удаляемый элемент, а затем выполнить merge для его сыновей l и r, и поставить результат объединения на место вершины t. Фактически, удаление из неявного декартова дерева не отличается от удаления из обычного декартова дерева.

- Сумма/минимум и т.п. на отрезке. Во-первых, для каждой вершины создадим дополнительное поле f в структуре item, в котором будет храниться значение целевой функции для поддерева этой вершины. Такое поле легко поддерживать, для этого надо поступить аналогично поддержке размеров cnt (создать функцию, вычисляющую значение этого поля, пользуясь его значениями для сыновей, и вставить вызовы этой функции в конце всех функций, меняющих дерево). Во-вторых, нам надо научиться отвечать на запрос на произвольном отрезке [A;B]. Научимся выделять из дерева его часть, соответствующую отрезку [A;B]. Нетрудно понять, что для этого достаточно сначала вызвать split (t, t1, t2, A), а затем split (t2, t2, t3, B-A+1). В результате дерево t2 и будет состоять из всех элементов в отрезке [A;B], и только них. Следовательно, ответ на запрос будет находиться в поле f вершины t2. После ответа на запрос дерево надо восстановить вызовами merge (t, t1, t2) и merge (t, t, t3).

- Прибавление/покраска на отрезке. Здесь мы поступаем аналогично предыдущему пункту, но вместо поля f будем хранить поле add, которое и будет содержать прибавляемую величину (или величину, в которую красят всё поддерево этой вершины). Перед выполнением любой операции эту величину add надо "протолкнуть т.е. соответствующим образом изменить t-l->add и t->r->add, а у себя значение add снять. Тем самым мы добьёмся того, что ни при каких изменениях дерева информация не будет потеряна.

- Переворот на отрезке. Этот пункт почти аналогичен предыдущему - нужно ввести поле bool rev, которое ставить в true, когда требуется произвести переворот в поддереве текущей вершины. "Проталкивание"поля rev заключается в том, что мы обмениваем местами сыновья текущей вершины, и ставим этот флаг для них.

```
void merge (pitem & t, pitem l, pitem r) {
    if (!l || !r)
        t = l ? l : r;
    else if (l->prior > r->prior)
        merge (l->r, l->r, r),  t = l;
    else
        merge (r->l, l, r->l),  t = r;
    upd_cnt (t);
}

void split (pitem t, pitem & l, pitem & r, int key, int add = 0) {
    if (!t)
        return void( l = r = 0 );
    int cur_key = add + cnt(t->l); // вычисляем неявный ключ
    if (key <= cur_key)
```

```
            split (t->l, l, t->l, key, add),  r = t;
        else
            split (t->r, t->r, r, key, add + 1 + cnt(t->l)),  l = t;
        upd_cnt (t);
    }
```

6 GCD

```
    int gcd (int a, int b, int & x, int & y) {
        if (a == 0) {
            x = 0; y = 1;
            return b;
        }
        int x1, y1;
        int d = gcd (b%a, a, x1, y1);
        x = y1 - (b / a) * x1;
        y = x1;
        return d;
    }
```

7 Convex hull

```
    struct pt {
        double x, y;
    };

    bool cmp (pt a, pt b) {
        return a.x < b.x || a.x == b.x && a.y < b.y;
    }

    bool cw (pt a, pt b, pt c) {
        return a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y) < 0;
    }

    bool ccw (pt a, pt b, pt c) {
        return a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y) > 0;
    }

    void convex_hull (vector<pt> & a) {
        if (a.size() == 1)  return;
        sort (a.begin(), a.end(), &cmp);
        pt p1 = a[0],  p2 = a.back();
        vector<pt> up, down;
        up.push_back (p1);
        down.push_back (p1);
        for (size_t i=1; i<a.size(); ++i) {
            if (i==a.size()-1 || cw (p1, a[i], p2)) {
                while (up.size()>=2 && !cw (up[up.size()-2], up[up.size()-1], a[i]))
                    up.pop_back();
                up.push_back (a[i]);
            }
            if (i==a.size()-1 || ccw (p1, a[i], p2)) {
                while (down.size()>=2 && !ccw (down[down.size()-2], down[down.size()-1], a[i]))
                    down.pop_back();
                down.push_back (a[i]);
            }
        }
        a.clear();
        for (size_t i=0; i<up.size(); ++i)
            a.push_back (up[i]);
        for (size_t i=down.size()-2; i>0; --i)
            a.push_back (down[i]);
    }
```

8 Z-func

Пусть дана строка s длины n. Тогда Z-функция ("зет-функция") от этой строки — это массив длины n, i-ый элемент которого равен наибольшему числу символов, начиная с позиции i, совпадающих с первыми символами строки s.

Иными словами, z[i] — это наибольший общий префикс строки s и её i-го суффикса.

Первый элемент Z-функции, z[0], обычно считают неопределённым. В данной статье мы будем считать, что он равен нулю (хотя ни в алгоритме, ни в приведённой реализации это ничего не меняет).

```
    vector<int> z_function (string s) {
        int n = (int) s.length();
```

```
    vector<int> z (n);
    for (int i=1, l=0, r=0; i<n; ++i) {
        if (i <= r)
            z[i] = min (r-i+1, z[i-l]);
        while (i+z[i] < n && s[z[i]] == s[i+z[i]])
            ++z[i];
        if (i+z[i]-1 > r)
            l = i,  r = i+z[i]-1;
    }
    return z;
}
```

9 Prefix-func

Дана строка $s[0 \dots n-1]$. Требуется вычислить для неё префикс-функцию, т.е. массив чисел $\pi[0 \dots n-1]$, где $\pi[i]$ определяется следующим образом: это такая наибольшая длина наибольшего собственного суффикса подстроки $s[0 \dots i]$, совпадающего с её префиксом (собственный суффикс — значит не совпадающий со всей строкой). В частности, значение $\pi[0]$ полагается равным нулю.

Пример — для строки "aabaaab"она равна: [0, 1, 0, 1, 2, 2, 3].

```
vector<int> prefix_function (string s) {
    int n = (int) s.length();
    vector<int> pi (n);
    for (int i=1; i<n; ++i) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j])  ++j;
        pi[i] = j;
    }
    return pi;
}
```

Алгоритм Кнута-Морриса-Пратта.

Эта задача является классическим применением префикс-функции (и, собственно, она и была открыта в связи с этим).

Дан текст t и строка s, требуется найти и вывести позиции всех вхождений строки s в текст t.

Обозначим для удобства через n длину строки s, а через m — длину текста t.

Образуем строку s + # + t, где символ # — это разделитель, который не должен нигде более встречаться. Посчитаем для этой строки префикс-функцию. Теперь рассмотрим её значения, кроме первых n+1 (которые, как видно, относятся к строке s и разделителю). По определению, значение $\pi[i]$ показывает наидлиннейшую длину подстроки, оканчивающейся в позиции i и совпадающего с префиксом. Но в нашем случае это $\pi[i]$ — фактически длина наибольшего блока совпадения со строкой s и оканчивающегося в позиции i. Больше, чем n, эта длина быть не может — за счёт разделителя. А вот равенство $\pi[i] = $ n (там, где оно достигается), означает, что в позиции i оканчивается искомое вхождение строки s (только не надо забывать, что все позиции отсчитываются в склеенной строке s+#+t).

Таким образом, если в какой-то позиции i оказалось $\pi[i] = $ n, то в позиции i - (n + 1) - n + 1 = i - 2 n строки t начинается очередное вхождение строки s в строку t.

Как уже упоминалось при описании алгоритма вычисления префикс-функции, если известно, что значения префикс-функции не будут превышать некоторой величины, то достаточно хранить не всю строку и префикс-функцию, а только её начало. В нашем случае это означает, что нужно хранить в памяти лишь строку s + # и значение префикс-функции на ней, а потом уже считывать по одному символу строку t и пересчитывать текущее значение префикс-функции.

10

```
struct vertex {
    int next[K];
    bool leaf;
    int p;
    char pch;
    int link;
    int go[K];
};

vertex t[NMAX+1];
int sz;
```

```

    void init() {
        t[0].p = t[0].link = -1;
        memset (t[0].next, 255, sizeof t[0].next);
        memset (t[0].go, 255, sizeof t[0].go);
        sz = 1;
    }

    void add_string (const string & s) {
        int v = 0;
        for (size_t i=0; i<s.length(); ++i) {
            char c = s[i]-'a';
            if (t[v].next[c] == -1) {
                memset (t[sz].next, 255, sizeof t[sz].next);
                memset (t[sz].go, 255, sizeof t[sz].go);
                t[sz].link = -1;
                t[sz].p = v;
                t[sz].pch = c;
                t[v].next[c] = sz++;
            }
            v = t[v].next[c];
        }
        t[v].leaf = true;
    }

    int go (int v, char c);

    int get_link (int v) {
        if (t[v].link == -1)
            if (v == 0 || t[v].p == 0)
                t[v].link = 0;
            else
                t[v].link = go (get_link (t[v].p), t[v].pch);
        return t[v].link;
    }

    int go (int v, char c) {
        if (t[v].go[c] == -1)
            if (t[v].next[c] != -1)
                t[v].go[c] = t[v].next[c];
            else
                t[v].go[c] = v==0 ? 0 : go (get_link (v), c);
        return t[v].go[c];
    }
```

11 Fast LCA

Воспользуемся классическим сведением задачи LCA к задаче RMQ (минимум на отрезке) (более подробно см. Наименьший общий предок. Нахождение за O (sqrt (N)) и O (log N) с препроцессингом O (N)). Научимся теперь решать задачу RMQ в данном частном случае с препроцессингом O (N) и O (1) на запрос.

Заметим, что задача RMQ, к которой мы свели задачу LCA, является весьма специфичной: любые два соседних элемента в массиве отличаются ровно на единицу (поскольку элементы массива - это не что иное как высоты вершин, посещаемых в порядке обхода, и мы либо идём в потомка, тогда следующий элемент будет на 1 больше, либо идём в предка, тогда следующий элемент будет на 1 меньше). Собственно алгоритм Фарах-Колтона и Бендера как раз и представляет собой решение такой задачи RMQ.

Обозначим через A массив, над которым выполняются запросы RMQ, а N - размер этого массива.

Построим сначала алгоритм, решающий эту задачу с препроцессингом O (N log N) и O (1) на запрос. Это сделать легко: создадим так называемую Sparse Table T[l,i], где каждый элемент T[l,i] равен минимуму A на промежутке [l; l+2i). Очевидно, $0 \leq i \leq \lceil \log N \rceil$, и потому размер Sparse Table будет O (N log N). Построить её также легко за O (N log N), если заметить, что T[l,i] = min (T[l,i-1], T[l+2i-1,i-1]). Как теперь отвечать на каждый запрос RMQ за O (1)? Пусть поступил запрос (l,r), тогда ответом будет min (T[l,sz], T[r-2sz+1,sz]), где sz - наибольшая степень двойки, не превосходящая r-l+1. Действительно, мы как бы берём отрезок (l,r) и покрываем его двумя отрезками длины 2sz - один начинающийся в l, а другой заканчивающийся в r (причём эти отрезки перекрываются, что в данном случае нам нисколько не мешает). Чтобы действительно достигнуть асимптотики O (1) на запрос, мы должны предпосчитать значения sz для всех возможных длин от 1 до N.

Теперь опишем, как улучшить этот алгоритм до асимптотики O (N).

Разобьём массив A на блоки размером K = 0.5 log2 N. Для каждого блока посчитаем минимальный элемент в нём и его позицию (поскольку для решения задачи LCA нам важны не сами минимумы, а их позиции). Пусть B - это массив размером N / K, составленный из этих минимумов в каждом блоке. Построим по массиву B Sparse Table, как описано выше, при этом размер Sparse Table и время её построения будут равны:

$$\frac{N}{K}\log\frac{N}{K} = \frac{2N}{\log N}\log\frac{2N}{\log N} = \frac{2N}{\log N}(1+\log\frac{N}{\log N}) \leq \frac{2N}{\log N} + 2N = O(N)$$

Теперь нам осталось только научиться быстро отвечать на запросы RMQ внутри каждого блока. В самом деле, если поступил запрос RMQ(l,r), то, если l и r находятся в разных блоках, то ответом будет минимум из следующих значений: минимум в блоке l, начиная с l и до конца блока, затем минимум в блоках после l и до r (не включительно), и наконец минимум в блоке r, от начала блока до r. На запрос "минимум в блоках"мы уже можем отвечать за O (1) с помощью Sparse Table, остались только запросы RMQ внутри блоков.

Здесь мы воспользуемся "+-1 свойством". Заметим, что, если внутри каждого блока от каждого его элемента отнять первый элемент, то все блоки будут однозначно определяться последовательностью длины K-1, состоящей из чисел +-1. Следовательно, количество различных блоков будет равно:

2K-1 = 20.5 log N - 1 = 0.5 sqrt(N) Итак, количество различных блоков будет O (sqrt (N)), и потому мы можем предпосчитать результаты RMQ внутри всех различных блоков за O (sqrt(N) K2) = O (sqrt(N) log2 N) = O (N). С точки зрения реализации, мы можем каждый блок характеризовать битовой маской длины K-1 (которая, очевидно, поместится в стандартный тип int), и хранить предпосчитанные RMQ в некотором массиве R[mask,l,r] размера O (sqrt(N) log2 N).

Итак, мы научились предпосчитывать результаты RMQ внутри каждого блока, а также RMQ над самими блоками, всё в сумме за O (N), а отвечать на каждый запрос RMQ за O (1) - пользуясь только предвычисленными значениями, в худшем случае четырьмя: в блоке l, в блоке r, и на блоках между l и r не включительно.

Реализация В начале программы указаны константы MAXN, LOG_MAXLIST и SQRT_MAXLIST, определяющие максимальное число вершин в графе, которые при необходимости надо увеличить.

```
const int MAXN = 100*1000;
const int MAXLIST = MAXN * 2;
const int LOG_MAXLIST = 18;
const int SQRT_MAXLIST = 447;
const int MAXBLOCKS = MAXLIST / ((LOG_MAXLIST+1)/2) + 1;

int n, root;
vector<int> g[MAXN];
int h[MAXN]; // vertex height
vector<int> a; // dfs list
int a_pos[MAXN]; // positions in dfs list
int block; // block size = 0.5 log A.size()
int bt[MAXBLOCKS][LOG_MAXLIST+1]; // sparse table on blocks (relative minimum positions in blocks)
int bhash[MAXBLOCKS]; // block hashes
int brmq[SQRT_MAXLIST][LOG_MAXLIST/2][LOG_MAXLIST/2]; // rmq inside each block, indexed by block hash
int log2[2*MAXN]; // precalced logarithms (floored values)

// walk graph
void dfs (int v, int curh) {
    h[v] = curh;
    a_pos[v] = (int)a.size();
    a.push_back (v);
    for (size_t i=0; i<g[v].size(); ++i)
        if (h[g[v][i]] == -1) {
            dfs (g[v][i], curh+1);
            a.push_back (v);
        }
}

int log (int n) {
    int res = 1;
    while (1<<res < n)  ++res;
    return res;
}

// compares two indices in a
inline int min_h (int i, int j) {
    return h[a[i]] < h[a[j]] ? i : j;
}

// O(N) preprocessing
void build_lca() {
    int sz = (int)a.size();
    block = (log(sz) + 1) / 2;
    int blocks = sz / block + (sz % block ? 1 : 0);

    // precalc in each block and build sparse table
    memset (bt, 255, sizeof bt);
    for (int i=0, bl=0, j=0; i<sz; ++i, ++j) {
        if (j == block)
            j = 0,  ++bl;
```

```
            if (bt[bl][0] == -1 || min_h (i, bt[bl][0]) == i)
                bt[bl][0] = i;
        }
        for (int j=1; j<=log(sz); ++j)
            for (int i=0; i<blocks; ++i) {
                int ni = i + (1<<(j-1));
                if (ni >= blocks)
                    bt[i][j] = bt[i][j-1];
                else
                    bt[i][j] = min_h (bt[i][j-1], bt[ni][j-1]);
            }

        // calc hashes of blocks
        memset (bhash, 0, sizeof bhash);
        for (int i=0, bl=0, j=0; i<sz||j<block; ++i, ++j) {
            if (j == block)
                j = 0,  ++bl;
            if (j > 0 && (i >= sz || min_h (i-1, i) == i-1))
                bhash[bl] += 1<<(j-1);
        }

        // precalc RMQ inside each unique block
        memset (brmq, 255, sizeof brmq);
        for (int i=0; i<blocks; ++i) {
            int id = bhash[i];
            if (brmq[id][0][0] != -1)  continue;
            for (int l=0; l<block; ++l) {
                brmq[id][l][l] = l;
                for (int r=l+1; r<block; ++r) {
                    brmq[id][l][r] = brmq[id][l][r-1];
                    if (i*block+r < sz)
                        brmq[id][l][r] =
                            min_h (i*block+brmq[id][l][r], i*block+r) - i*block;
                }
            }
        }

        // precalc logarithms
        for (int i=0, j=0; i<sz; ++i) {
            if (1<<(j+1) <= i)  ++j;
            log2[i] = j;
        }
    }

    // answers RMQ in block #bl [l;r] in O(1)
    inline int lca_in_block (int bl, int l, int r) {
        return brmq[bhash[bl]][l][r] + bl*block;
    }

    // answers LCA in O(1)
    int lca (int v1, int v2) {
        int l = a_pos[v1],  r = a_pos[v2];
        if (l > r)  swap (l, r);
        int bl = l/block,  br = r/block;
        if (bl == br)
            return a[lca_in_block(bl,l%block,r%block)];
        int ans1 = lca_in_block(bl,l%block,block-1);
        int ans2 = lca_in_block(br,0,r%block);
        int ans = min_h (ans1, ans2);
        if (bl < br - 1) {
            int pw2 = log2[br-bl-1];
            int ans3 = bt[bl+1][pw2];
            int ans4 = bt[br-(1<<pw2)][pw2];
            ans = min_h (ans, min_h (ans3, ans4));
        }
        return a[ans];
    }
```

12 Geom 1

Поиск пары пересекающихся отрезков за O (N log N)

```
    const double EPS = 1E-9;

    struct pt {
        double x, y;
    };

    struct seg {
        pt p, q;
        int id;

```

```
        double get_y (double x) const {
            if (abs (p.x - q.x) < EPS)  return p.y;
            return p.y + (q.y - p.y) * (x - p.x) / (q.x - p.x);
        }
    };


    inline bool intersect1d (double l1, double r1, double l2, double r2) {
        if (l1 > r1)  swap (l1, r1);
        if (l2 > r2)  swap (l2, r2);
        return max (l1, l2) <= min (r1, r2) + EPS;
    }

    inline int vec (const pt & a, const pt & b, const pt & c) {
        double s = (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
        return abs(s)<EPS ? 0 : s>0 ? +1 : -1;
    }

    bool intersect (const seg & a, const seg & b) {
        return intersect1d (a.p.x, a.q.x, b.p.x, b.q.x)
            && intersect1d (a.p.y, a.q.y, b.p.y, b.q.y)
            && vec (a.p, a.q, b.p) * vec (a.p, a.q, b.q) <= 0
            && vec (b.p, b.q, a.p) * vec (b.p, b.q, a.q) <= 0;
    }


    bool operator< (const seg & a, const seg & b) {
        double x = max (min (a.p.x, a.q.x), min (b.p.x, b.q.x));
        return a.get_y(x) < b.get_y(x) - EPS;
    }


    struct event {
        double x;
        int tp, id;

        event() { }
        event (double x, int tp, int id)
            : x(x), tp(tp), id(id)
        { }

        bool operator< (const event & e) const {
            if (abs (x - e.x) > EPS)  return x < e.x;
            return tp > e.tp;
        }
    };

    set<seg> s;
    vector < set<seg>::iterator > where;

    inline set<seg>::iterator prev (set<seg>::iterator it) {
        return it == s.begin() ? s.end() : --it;
    }

    inline set<seg>::iterator next (set<seg>::iterator it) {
        return ++it;
    }

    pair<int,int> solve (const vector<seg> & a) {
        int n = (int) a.size();
        vector<event> e;
        for (int i=0; i<n; ++i) {
            e.push_back (event (min (a[i].p.x, a[i].q.x), +1, i));
            e.push_back (event (max (a[i].p.x, a[i].q.x), -1, i));
        }
        sort (e.begin(), e.end());

        s.clear();
        where.resize (a.size());
        for (size_t i=0; i<e.size(); ++i) {
            int id = e[i].id;
            if (e[i].tp == +1) {
                set<seg>::iterator
                    nxt = s.lower_bound (a[id]),
                    prv = prev (nxt);
                if (nxt != s.end() && intersect (*nxt, a[id]))
                    return make_pair (nxt->id, id);
                if (prv != s.end() && intersect (*prv, a[id]))
                    return make_pair (prv->id, id);
                where[id] = s.insert (nxt, a[id]);
            }
            else {
                set<seg>::iterator
                    nxt = next (where[id]),
```

```
                prv = prev (where[id]);
            if (nxt != s.end() && prv != s.end() && intersect (*nxt, *prv))
                return make_pair (prv->id, nxt->id);
            s.erase (where[id]);
        }
    }

    return make_pair (-1, -1);
}
```

13 Discrete Fourier Transform

Пусть имеется многочлен n-ой степени:

$$A(x) = a_0x^0 + a_1x^1 + \ldots + a_{n-1}x^{n-1}$$

Не теряя общности, можно считать, что n является степенью 2. Если в действительности n не является степенью 2, то мы просто добавим недостающие коэффициенты, положив их равными нулю.

Из теории функций комплексного переменного известно, что комплексных корней n-ой степени из единицы существует ровно n. Обозначим эти корни через $w_{n,k}, k = 0 \ldots n-1$, тогда известно, что $w_{n,k} = e^{i\frac{2\pi k}{n}}$. Кроме того, один из этих корней $w_n = w_{n,1} = e^{i\frac{2\pi}{n}}$ (называемый главным значением корня n-ой степени из единицы) таков, что все остальные корни являются его степенями: $w_{n,k} = (w_n)^k$.

Тогда дискретным преобразованием Фурье (ДПФ) (discrete Fourier transform, DFT) многочлена A(x) (или, что то же самое, ДПФ вектора его коэффициентов $(a_0, a_1, \ldots, a_{n-1})$) называются значения этого многочлена в точках $x = w_{n,k}$, т.е. это вектор:

$$\mathrm{DFT}(a_0, a_1, \ldots, a_{n-1}) = (y_0, y_1, ..., y_{n-1}) = (A(w_n^0), A(w_n^1), \ldots, A(w_n^{n-1})).$$

Аналогично определяется и обратное дискретное преобразование Фурье (InverseDFT). Обратное ДПФ для вектора значений многочлена $(y_0, y_1, \ldots y_{n-1})$ — это вектор коэффициентов многочлена $(a_0, a_1, \ldots, a_{n-1})$:

$$\mathrm{InverseDFT}(y_0, y_1, \ldots, y_{n-1}) = (a_0, a_1, ..., a_{n-1})$$

Таким образом, если прямое ДПФ переходит от коэффициентов многочлена к его значениям в комплексных корнях n-ой степени из единицы, то обратное ДПФ — наоборот, по значениям многочлена восстанавливает коэффициенты многочлена.

```
const int mod = 7340033;
const int root = 5;
const int root_1 = 4404020;
const int root_pw = 1<<20;

void fft (vector<int> & a, bool invert) {
    int n = (int) a.size();

    for (int i=1, j=0; i<n; ++i) {
        int bit = n >> 1;
        for (; j>=bit; bit>>=1)
            j -= bit;
        j += bit;
        if (i < j)
            swap (a[i], a[j]);
    }

    for (int len=2; len<=n; len<<=1) {
        int wlen = invert ? root_1 : root;
        for (int i=len; i<root_pw; i<<=1)
            wlen = int (wlen * 1ll * wlen % mod);
        for (int i=0; i<n; i+=len) {
            int w = 1;
            for (int j=0; j<len/2; ++j) {
                int u = a[i+j],  v = int (a[i+j+len/2] * 1ll * w % mod);
                a[i+j] = u+v < mod ? u+v : u+v-mod;
                a[i+j+len/2] = u-v >= 0 ? u-v : u-v+mod;
                w = int (w * 1ll * wlen % mod);
            }
        }
    }
    if (invert) {
        int nrev = reverse (n, mod);
        for (int i=0; i<n; ++i)
```

```
            a[i] = int (a[i] * 1ll * nrev % mod);
        }
}

void multiply (const vector<int> & a, const vector<int> & b, vector<int> & res) {
    vector<base> fa (a.begin(), a.end()),  fb (b.begin(), b.end());
    size_t n = 1;
    while (n < max (a.size(), b.size()))  n <<= 1;
    n <<= 1;
    fa.resize (n),  fb.resize (n);

    fft (fa, false),  fft (fb, false);
    for (size_t i=0; i<n; ++i)
        fa[i] *= fb[i];
    fft (fa, true);

    res.resize (n);
    for (size_t i=0; i<n; ++i)
        res[i] = int (fa[i].real() + 0.5);
}
```

14 Graph

Алгоритм Диница нахождения максимального потока за $O(N^2M)$

```
const int MAXN = ...; // число вершин
const int INF = 1000000000; // константа-бесконечность

int n, c[MAXN][MAXN], f[MAXN][MAXN], s, t, d[MAXN], ptr[MAXN], q[MAXN];

bool bfs() {
    int qh=0, qt=0;
    q[qt++] = s;
    memset (d, -1, n * sizeof d[0]);
    d[s] = 0;
    while (qh < qt) {
        int v = q[qh++];
        for (int to=0; to<n; ++to)
            if (d[to] == -1 && f[v][to] < c[v][to]) {
                q[qt++] = to;
                d[to] = d[v] + 1;
            }
    }
    return d[t] != -1;
}

int dfs (int v, int flow) {
    if (!flow)  return 0;
    if (v == t)  return flow;
    for (int & to=ptr[v]; to<n; ++to) {
        if (d[to] != d[v] + 1)  continue;
        int pushed = dfs (to, min (flow, c[v][to] - f[v][to]));
        if (pushed) {
            f[v][to] += pushed;
            f[to][v] -= pushed;
            return pushed;
        }
    }
    return 0;
}

int dinic() {
    int flow = 0;
    for (;;) {
        if (!bfs())  break;
        memset (ptr, 0, n * sizeof ptr[0]);
        while (int pushed = dfs (s, INF))
            flow += pushed;
    }
    return flow;
}
```