```
// CMAKE
// ********************************
/*
cmake_minimum_required(VERSION 3.22)
project(olymp)

set(CMAKE_CXX_STANDARD 20)

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Wshadow -g -fsanitize=
    undefined -fsanitize=bounds -fsanitize=address -D_GLIBCXX_DEBUG")

add_executable(olymp
        main.cc)

add_compile_definitions(LOCAL=true)
*/
// ********************************


// STRESS-tests
// ********************************

// SHELL-client
/*
out_data="a"
out_sol_data="a"
tested=0
while [ "$out_data" = "$out_sol_data" ]
do
    in_data="$(python3 "gen.py")"
    out_data="$(echo $in_data | ../cmake-build-debug/olymp)"
    out_sol_data="$(echo $in_data | ./solution)"
    ((tested++))
    if [ "$(expr $tested % 100)" = "0" ]
    then
      echo -e "TESTING IN PROGRESS...\nCASES TESTED: $tested" > "log"
    fi
done
if [ "$1" = "-f" ]
then
    echo -e "TESTING FOUND MISMATCH.\nCASES TESTED: $tested\n" > "log"
    echo "$in_data" > "in"
    echo "$out_data" > "out"
    echo "$out_sol_data" > "out_sol"
else
    echo -e "TESTING FOUND MISMATCH.\nCASES TESTED: $tested\n\nINPUT:\
     n$in_data\n\n"
    echo -e "OUTPUT:\n$out_data\n\nSOLUTION OUTPUT:\n$out_sol_data\n"
fi
*/


// GENERATORS (python) by github.com/TheEvilBird
/*
# files: generators.py gen.py stress.py

import random
import heapq

def gen_num(L: int, R: int):
    """
    Generates a number between L and R.
    """
    return random.randint(L, R)

# abcdefghijklmnopqrstuvwxyz

def gen_string_abc(LEN: int, ALPH_LEN: int = 26):
    """
    Generates a string of length LEN using the first ALPH_LEN lowercase
     letters of the alphabet.
    """
    abc = "abcdefghijklmnopqrstuvwxyz"
    s = abc[:ALPH_LEN]
    res = ""
    for i in range(LEN):
        res += random.choice(s)
    return res

def gen_string_any_aplh(LEN: int, ALPH: str):
    """
    Generates a string of length LEN using ALPH as the alphabet.
    """
    res = ""
    # ALPH_LEN = len(ALPH)
    for i in range(LEN):
        kek = 1
        res += random.choice(ALPH)
    return res

def gen_tree(N: int):
    """
    Generates a tree with N vertices.
    """
    edges = []
    for i in range(2, N + 1):
        v = gen_num(1, i - 1)
        edges.append((v, i))
    return edges

def gen_DAG(N: int, M: int):
    """
    Generates a directed acyclic graph with N vertices and M edges.
    """
    edges = []
    while len(edges) < M:
        v = gen_num(1, N - 1)
        u = gen_num(v + 1, N)
        edges.append((v, u))
    return edges

def gen_graph(N: int, M: int):
    """
    Generates a graph with N vertices and M edges.
    """
    edges_set = set()
    for i in range(M):
        v, u = 0, 0
        while (v, u) in edges_set or v == u:
            v, u = gen_num(1, N), gen_num(1, N)
            v, u = min(v, u), max(v, u)
        edges_set.add((v, u))
    return list(edges_set)

def gen_multigraph(N: int, M: int):
    """
    Generates a multigraph with N vertices and M edges.
    """
    edges = []
    for i in range(M):
        v, u = -1, 0
        while v == -1:
            v, u = gen_num(1, N), gen_num(1, N)
            v, u = min(v, u), max(v, u)
        edges.append((v, u))
    return edges

def gen_directed_graph(N: int, M: int):
    """
    Generates a directed graph with N vertices and M edges.
    """
    edges_set = set()
    for i in range(M):
        v, u = 0, 0
        while (v, u) in edges_set or v == u:
            v, u = gen_num(1, N), gen_num(1, N)
        edges_set.add((v, u))
    return list(edges_set)

def gen_connected_directed_graph(N: int, M: int):
    """
    Generates a directed connected graph with N vertices and M edges.
    """
    edges_set = set(gen_tree(N))
    for i in range(M - (N - 1)):
        v, u = 0, 0
        while (v, u) in edges_set or v == u:
            v, u = gen_num(1, N), gen_num(1, N)
        edges_set.add((v, u))
    return list(edges_set)

def gen_connected_graph(N: int, M: int):
    """
    Generates a connected graph with N vertices and M edges.
    """
    edges_set = set(gen_tree(N))
    for i in range(M - (N - 1)):
        v, u = 0, 0
        while (v, u) in edges_set or v == u:
            v, u = gen_num(1, N), gen_num(1, N)
            v, u = min(v, u), max(v, u)
        edges_set.add((v, u))
    return list(edges_set)

def gen_connected_multigraph(N: int, M: int):
    """
    Generates a connected multigraph with N vertices and M edges.
    """
    edges = gen_tree(N)
    for i in range(M - (N - 1)):
        v, u = 0, 0
        while v == u:
            v, u = gen_num(1, N), gen_num(1, N)
        edges.append((v, u))
    return edges

def gen_perm(N: int, FIR: int = 1):
    """
    Generates a permutation of length N with min element FIR.
    """
    arr = [FIR + i for i in range(N)]
    # arr = arr[1:]
    random.shuffle(arr)
    return arr
```

```
def gen_array(N: int, L: int, R: int):
    """
    Generates an array of length N with elements between L and R.
    """
    arr = [gen_num(L, R) for i in range(N)]
    return arr

def gen_array_pairs(N: int, L: int, R: int):
    """
    Generates an array of pairs of length N with elements between L and R
      .
    """
    arr = [(gen_num(L, R), gen_num(L, R)) for i in range(N)]
    return arr

def gen_array_pairs(N: int, L1: int, R1: int, L2: int, R2: int):
    """
    Generates an array of pairs of length N with the first elements of
      each pair between L1 and R1 and between L2 and R2 for the second
      element.
    """
    arr = [(gen_num(L1, R1), gen_num(L2, R2)) for i in range(N)]
    return arr
*/

// ********************************


#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

#ifndef LOCAL
#pragma GCC optimize("O3,unroll-loops")
    #pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
#endif

using namespace std;
using namespace __gnu_pbds;
template<class T> // order_of_key, find_by_order
using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
        tree_order_statistics_node_update>;

#define int int64_t
#define float double_t
#define cmpl complex<float>

#define pii pair<int, int>
#define pff pair<float, float>
#define pcc pair<cmpl, cmpl>
#define i1 first
#define i2 second

#define LINF 1'000'000'000'000'000'001
#define INF 1'000'000'001
#define EPS 1e-8
#define MOD0 1'000'000'007
#define MOD1 998'244'353
#define MOD2 1'000'000'483
#define MOD3 129'061
#define MOD4 3'000'061
#define P0 257
#define P1 283
#define P2 293

mt19937 global_rnd{};


// NUMBER THEORY
// ********************************

int bin_pow(int n, int p) { /**    n*m = 1 (mod p)  =>  m = n**(p-2) (mod
       p)    **/
    if (p == 0) return 1;
    int nn = bin_pow(n, p/2);
    if (p&1) return n*nn*nn;
    return nn*nn;
}

int bin_pow(int n, int p, int mod) { /**    n*m = 1 (mod p)  =>  m = n**(
      p-2) (mod p)    **/
    if (p == 0) return 1;
    int nn = bin_pow(n, p/2)%mod;
    if (p&1) return (((n*nn)%mod)*nn)%mod;
    return (nn*nn)%mod;
}


// extended gcd / diophantines / reversed element
int ext_gcd(int a, int b, int& x, int& y) {
    if (a < b)
        return ext_gcd(b, a, y, x);
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    int x1, y1;
    int g = ext_gcd(b, a%b, x1, y1);
    x = y1;
    y = x1 - (a/b)*y1;
    return g;
}

// LINEAR SIEVE
// multiplicative arithmetic functions calculation in [1..n]
vector<int> sieve(int n) {
    int lp[n+1];
    vector<int> pr;

    for (int i = 2; i <= n; ++i) {
        if (lp[i] == 0) {
            lp[i] = i;
            pr.push_back (i);
        }

        for (int j = 0; j < (int)pr.size() && pr[j] <= lp[i] && i*pr[j]
   <= n; ++j)
            lp[i * pr[j]] = pr[j];
    }

    return pr;
}

// FACTORIZATION: Pollard's Ro-algorithm
int find_divisor(int n, int seed = 1) {
    auto f = [](int x, int n)
            { return (__int128_t) (x + 1) * (x + 1) % n; };
    int x = seed, y = seed;
    int d = 1;
    while (d == 1 || d == n) {
        y = f(y, n);
        x = f(f(x, n), n);
        d = gcd(abs(x - y), n);
    }
    return d;
}



// DFT: FAST FOURIER TRANSFORM (polynomial)
void fft(vector<cmpl>& a, bool invert) {
    int n = a.size(), h = -1;
    vector<int> rev(n, 0);
    for (int i = 1; i < n; ++i) {
        if (!(i & (i - 1)))
            ++h;
        rev[i] = rev[i ^ (1 << h)] | (1 << (__lg(n) - 1 - h));
    }
    for (int i = 0; i < n; ++i) {
        if (i < rev[i])
            swap(a[i], a[rev[i]]);
    }

    double alpha = 2 * M_PI / n * (invert ? -1 : 1);
    cmpl w1(cos(alpha), sin(alpha));
    vector<cmpl> W(n >> 1, 1);
    for (int i = 1; i < (n >> 1); ++i)
        W[i] = W[i - 1] * w1;

    for (int i = 0; i < __lg(n); ++i)
        for (int j = 0; j < n; ++j)
            if (!(j & (1 << i))) {
                cmpl t = a[j ^ (1 << i)] * W[(j & ((1 << i) - 1)) * (n >>
      (i + 1))];
                a[j ^ (1 << i)] = a[j] - t;
                a[j] = a[j] + t;
            }

    if (invert)
        for (int i = 0; i < n; i++)
            a[i] /= n;
}

void mul(const vector<float>& a, const vector<float>& b, vector<float>&
     res) {
    int n = 1;
    while (n < a.size() || n < b.size())
        n <<= 1;
    n <<= 1;
    vector<cmpl> dft_a(a.begin(), a.end());
    vector<cmpl> dft_b(b.begin(), b.end());
    dft_a.resize(n);
    dft_b.resize(n);
    fft(dft_a, false);
    fft(dft_b, false);
    for (int i = 0; i < n; i++)
        dft_a[i] *= dft_b[i];
    fft(dft_a, true);
    res.resize(n);
    for (int i = 0; i < n; i++)
        res[i] = dft_a[i].real();
//        res[i] = (int)(dft_a[i].real() + 0.1);
```

```cpp
}


// LINEAR SYSTEMS
int gauss(vector<vector<double>> a, vector<double> &ans) {
    int n = (int)a.size(), m = (int)a[0].size() - 1;
    vector<int> pos(m, -1);
    double det = 1; int rank = 0;
    for(int col = 0, row = 0; col < m && row < n; ++col) {
        int mx = row;
        for (int i = row; i < n; i++) {
            if (fabs(a[i][col]) > fabs(a[mx][col])) { mx = i; }
        }
        if (fabs(a[mx][col]) < EPS) { det = 0; continue; }
        for (int i = col; i <= m; i++) {
            swap(a[row][i], a[mx][i]);
        }
        if (row != mx) { det = -det; }
        det *= a[row][col];
        pos[col] = row;
        for (int i = 0; i < n; i++) {
            if (i != row && fabs(a[i][col]) > EPS) {
                double c = a[i][col] / a[row][col];
                for (int j = col; j <= m; j++) {
                    a[i][j] -= a[row][j] * c;
                }
            }
        }
        ++row; ++rank;
    }
    ans.assign(m, 0);
    for(int i = 0; i < m; i++) {
        if (pos[i] != -1) { ans[i] = a[pos[i]][m] / a[pos[i]][i]; }
    }
    for(int i = 0; i < n; i++) {
        double sum = 0;
        for (int j = 0; j < m; j++) {
            sum += ans[j] * a[i][j];
        }
        if (fabs(sum - a[i][m]) > EPS) {
            return -1; //no solution
        }
    }
    for (int i = 0; i < m; i++) {
        if (pos[i] == -1) {
            return 2; //infinte solutions
        }
    }
    return 1; //unique solution
}

// ********************************



// Dynamic Programming, bitmasks
// ********************************

// submasks O(3^n)
void submasks(int n) {
    for (int m = 0; m < (1 << n); ++m)
        for (int s = m; s; s = (s - 1) & m);
}

// SOS DP
vector<int> sos_dp(vector<int>& A, int N) {
    vector<int> F(N);
    for (int i = 0; i < (1 << N); ++i)
        F[i] = A[i];
    for (int i = 0; i < N; ++i)
        for (int mask = 0; mask < (1 << N); ++mask) {
            if (mask & (1 << i))
                F[mask] += F[mask ^ (1 << i)];
        }
    return F;
}


// LEVENSTEIN DISTANCE
int levenstein_dist(string& s, string& t) {
    int n = (int)s.size(), m = (int)t.size();
    vector<vector<int>> dp(n+1, vector<int>(m+1, INF));
    dp[0][0] = 0;
    for (int i = 1; i <= n; ++i)
        dp[i][0] = i;
    for (int i = 1; i <= m; ++i)
        dp[0][i] = i;
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= m; ++j)
            dp[i][j] = min(min(dp[i-1][j]+1, dp[i][j-1]+1),
                           dp[i-1][j-1] + (s[i-1] != t[j-1]));
    return dp[n][m];
}

// ********************************

// STRINGS
// ********************************

vector<int> pi_func(const string& s) {
    int n = (int)s.size();
    vector<int> pi(n, 0);
    for (int i = 1; i < n; ++i) {
        int x = pi[i-1];
        while (x > 0 && s[x] != s[i])
            x = pi[x-1];
        pi[i] = x + (s[x] == s[i]);
    }
    return pi;
}

vector<int> z_func(const string& s) {
    int n = (int)s.size();
    vector<int> z(n, 0);
    z[0] = (int)s.size();
    int l = 0, r = 0;
    for (int i = 1; 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;
}

int manacher(const string& s) {
    int n = (int)s.size();
    vector<int> m(n, 1);
    int l = 0, r = 0, count = 0;
    for (int i = 1; i < n; ++i) {
        if (i <= r)
            m[i] = min(r-i+1, m[l+r-i]);
        while (i-m[i] >= 0 && i+m[i] < n && s[i-m[i]] == s[i+m[i]])
            ++m[i];
        if (i+m[i]-1 >= r) {
            l = i-m[i]+1;
            r = i+m[i]-1;
        }
        if (s[i] == '#') count += m[i]/2;
        else count += (m[i]+1)/2;
    }
    return count;
}



// HASHES
class polynomial_hash {
public:
    polynomial_hash(string* _s, int _p, int _m);
    int rh(int l, int r);
    bool verify(int l1, int r1, int l2, int r2);
private:
    string* s;
    vector<int> ps, hs;
    int n, p, m;
};

polynomial_hash::polynomial_hash(string* _s, int _p, int _m) : s(_s), p(
    _p), m(_m) {
    n = (int)s->size();
    ps = vector<int>(n+1, 1);
    for (int i = 1; i <= n; ++i)
        ps[i] = (p*ps[i-1]) % m;
    hs = vector<int>(n+1, 0);
    for (int i = 1; i <= n; ++i)
        hs[i] = (hs[i-1] + ps[i]*s->at(i-1)) % m;
}

int polynomial_hash::rh(int l, int r) {
    return (ps[n-l]*(hs[r+1] - hs[l] + m)) % m;
}

bool polynomial_hash::verify(int l1, int r1, int l2, int r2) {
    return rh(l1, r1) == rh(l2, r2);
}


bool double_hash_verify(polynomial_hash& f, polynomial_hash& s, int l1,
     int r1, int l2, int r2) {
    return f.verify(l1, r1, l2, r2) && s.verify(l1, r1, l2, r2);
}


int str_dif(string& s, polynomial_hash& h1, polynomial_hash& h2, int l1,
     int r1, int l2, int r2) {
    int d = -1;
    int l = 0, r = min(r1-l1, r2-l2);
    while (l <= r) {
        int m = l + (r-l)/2;
```

```cpp
        if (double_hash_verify(h1, h2, l1, l1 + m, l2, l2 + m)) {
            d = max(d, m);
            l = m+1;
            continue;
        }
        r = m-1;
    }
    return d;
}


struct pair_hash {
    template <class T1, class T2>
    size_t operator () (pair<T1, T2> const & pair) const {
        size_t h1 = hash<T1>()(pair.first);
        size_t h2 = hash<T2>()(pair.second);
        return h1 ^ h2;
    }
};

struct vector_hash {
    const size_t p1 = 283, p2 = 293, m1 = 0x34fd319f;

    template<class T>
    size_t operator () (vector<T> const & vec) const {
        const size_t sz2 = (sizeof(T) >> 1);
        size_t seed = vec.size();
        for (auto x : vec) {
            x ^= (x << sz2) + p1;
            x ^= (x >> sz2) * p2;
            seed ^= ((seed << sz2) + P0) ^ x;
            seed ^= (x >> 2) + (seed >> 3) + m1;
        }
        return seed;
    }
};


// SUFFIX STRUCTURES

void suffix_array(string s, vector<vector<int>>& ps, vector<vector<int>>&
     cs) {
    s.push_back(0);
    int n = (int)s.size(), cnt = 0, cls = 0;
    vector<int> p(n), c(n);

    map<char, vector<int>> t;
    for (int i = 0; i < n; ++i)
        t[s[i]].push_back(i);

    for (auto& [ch, ids] : t) {
        for (auto i : ids) {
            c[i] = cls;
            p[cnt++] = i;
        }
        ++cls;
    }

    ps.emplace_back(p.begin()+1, p.end());
    cs.emplace_back(c.begin(), prev(c.end()));

    for (int l = 1; cls < n; ++l) {
        int d = (1 << (l-1));
        int _cls = 0;
        cnt = 0;
        vector<int> _c(n);
        vector<vector<int>> a(n);

        for (int i = 0; i < n; ++i) {
            int k = (p[i] - d + n) % n;
            a[c[k]].push_back(k);
        }

        for (int i = 0; i < cls; ++i)
            for (int j = 0; j < a[i].size(); ++j) {
                if (j == 0 || c[(a[i][j] + d) % n] != c[(a[i][j-1] + d) %
      n])
                    ++_cls;
                _c[a[i][j]] = _cls-1;
                p[cnt++] = a[i][j];
            }

        c = _c;
        cls = _cls;

        ps.emplace_back(p.begin()+1, p.end());
        cs.emplace_back(c.begin(), prev(c.end()));
    }
}

// lcp of two suffixes O(logn)
int lcp_binups(vector<vector<int>>& c, int i, int j) {
    int res = 0;
    for (int k = (int)(c.size())-1; k >= 0; --k)
        if (c[k][i] == c[k][j]) {
            res += (1<<k);
            i += (1<<k);
            j += (1<<k);
        }
    return res;
}

// Kasai, Arimura, Arikawa, Lee, Park algorithm (lcp(i) := lcp(p[i], p[i
     +1])
vector<int> lcp_5(string& s, vector<int>& p, vector<int>& c) {
    int n = (int)s.size();
    int l = 0;
    vector<int> lcp(n, 0);
    for (int i = 0; i < n; ++i) {
        if (c[i] == n)
            continue;
        int nxt = p[c[i]];
        while (max(i, nxt)+l < n && s[i+l] == s[nxt+l])
            ++l;
        lcp[c[i]-1] = l;
        l = max(l-1, 0l);
    }
    return lcp;
}

// ********************************


// DSU
// ********************************

class dsu {
public:
    dsu(int n);
    int classify(int u);
    void unite(int u, int v);
private:
    vector<int> state;
    vector<int> h;
};

dsu::dsu(int n) {
    state.resize(n);
    h.resize(n, 0);
    for (int i = 0; i < n; ++i)
        state[i] = i;
}

int dsu::classify(int u) {
    if (state[u] == u)
        return u;
    return state[u] = this->classify(state[u]);
}

void dsu::unite(int u, int v) {
    u = this->classify(u);
    v = this->classify(v);
    if (h[u] >= h[v])
        state[v] = u;
    else
        state[u] = v;
    if (h[u] == h[v])
        ++h[u];
}

// ********************************


// GRAPHS
// ********************************

// DFS
void dfs(const vector<vector<int>>& g, vector<bool>& mark, vector<int>&
      tin, vector<int>& tout,
         int& tm, int u) {
    tin[u] = tm++;
    mark[u] = 1;
    for (auto& v : g[u]) {
        if (!mark[v]) {
            dfs(g, mark, tin, tout, tm, v);
        }
    }
    tout[u] = tm++;
}

// BFS
void bfs(const vector<vector<int>>& g, vector<int>& mark, int start) {
    vector<int> state;
    state.push_back(start);
    while (!state.empty()) {
        int u = state.back();
        state.pop_back();
        mark[u] = 1;
        for (auto& v : g[u])
                state.push_back(v);
    }
}
```

```cpp
// ALGS FOR SEARCHING SHORTEST WAYS

// DIJKSTRA + HEAP (SET)
vector<int> dijkstra(const vector<vector<pii>>& g, int start) {
    vector<int> dist(g.size(), INF);
    dist[start] = 0;
    set<pii> state;
    state.insert({0, start});
    while (!state.empty()) {
        auto first = state.begin();
        int u = first->second;
        state.erase(first);
        for (auto& [wt, v] : g[u])
            if (dist[u]+wt < dist[v]) {
                state.erase({dist[v], v});
                dist[v] = dist[u]+wt;
                state.insert({dist[v], v});
            }
    }
    return dist;
}

// FLOYD
void floyd(vector<vector<int>>& dp) {
    int n = dp.size();
    for (int u = 0; u < n; ++u)
        for (int v = 0; v < n; ++v)
            for (int k = 0; k < n; ++k)
                if (dp[u][k] < INF && dp[k][v] < INF && dp[u][v] > dp[u][
     k]+dp[k][k])
                    dp[u][v] = dp[u][k]+dp[k][v];
}

// FORD-BELLMAN
vector<int> ford_beintman(const vector<tuple<int, int, int>>& e, int n,
     int start) {
    vector<int> dist(n, INF);
    for (int i = 0; i < n; ++i)
        for (auto& [u, v, wt] : e)
            if (dist[u] < INF && dist[u]+wt < dist[v])
                dist[v] = dist[u]+wt;
    return dist;
}


// EULER GRAPHS
void euler_way(vector<unordered_multiset<int>>& g, int v, vector<int>&
     way) {
    while (!g[v].empty()) {
        auto u = *g[v].begin();
        g[v].erase(g[v].begin());
        g[u].erase(g[u].find(v));
        euler_way(g, u, way);
    }
    way.push_back(v);
}

// TOP_SORT
void top_sort(const vector<vector<int>>& g, vector<bool>& mark, vector<
     int>& top, int v) {
    mark[v] = 1;
    for (auto& u : g[v])
        if (!mark[u])
            top_sort(g, mark, top, u);
    top.push_back(v);
}


// STRONG COMPONENTS (COND.)
void strong_comp(const vector<vector<int>>& gr, vector<bool>& mark,
     vector<int>& comp, int v) {
    mark[v] = 1;
    comp.push_back(v);
    for (auto& u : gr[v])
        if (!mark[u])
            strong_comp(gr, mark, comp, u);
}

vector<vector<int>> build_strong_comps(int n, int m) {
    vector<vector<int>> g(n+1), gr(n+1);
    for (int i = 1; i <= m; ++i) {
        int u, v;
        cin >> u >> v;
        g[u].push_back(v);
        gr[v].push_back(u);
    }
    vector<bool> mark(n+1, 0);
    vector<int> top;
    for (int v = 1; v <= n; ++v)
        if (!mark[v])
            top_sort(g, mark, top, v);
    mark = vector<bool>(n+1, 0);
    vector<vector<int>> comps;
    int k = 0;
    for (int i = 0; i < n; ++i) {
        int v = top[n-1-i];
        if (!mark[v]) {
            comps.push_back(vector<int>());
            ++k;
            strong_comp(gr, mark, comps[k-1], v);
        }
    }
    return comps;
}


// k-connected graphs

// BRIDGES
void bridges(vector<vector<int>>& g, vector<bool>& mark, vector<int>& tin
     , vector<int>& dp, set<pii>& br,
             int& timer, int p, int v) {
    tin[v] = timer++;
    mark[v] = true;
    dp[v] = tin[v];
    for (auto& u : g[v]) {
        if (u == p) continue;
        if (mark[u]) {
            dp[v] = min(dp[v], tin[u]);
            br.erase({u, v});
            continue;
        }
        bridges(g, mark, tin, dp, br, timer, v, u);
        dp[v] = min(dp[v], dp[u]);
        if (dp[u] > tin[v] && u != v)
            br.emplace(u, v);
    }
}

// CUT POINTS
void cut_points(vector<vector<int>>& g, vector<bool>& mark, vector<int>&
     tin, vector<int>& dp,
           set<int>& cp, int& timer, int p, int v) {
    tin[v] = timer++;
    mark[v] = true;
    dp[v] = tin[v];
    int ch = 0;
    for (auto& u : g[v]) {
        if (u == p) continue;
        if (mark[u]) {
            dp[v] = min(dp[v], tin[u]);
            continue;
        }
        cut_points(g, mark, tin, dp, cp, timer, v, u);
        dp[v] = min(dp[v], dp[u]);
        if (tin[v] <= dp[u] && p != -1)
            cp.insert(v);
        ++ch;
    }
    if (p == -1 && ch > 1)
        cp.insert(v);
}


// MST

// O(|V|**2) for dense graph G=(V, E): |E| ~ |V|**2
vector<vector<int>> prim(vector<vector<int>>& g, int start) { // "g" is
     adjacent matrix
    int n = (int)g.size();
    vector<bool> mark(n, false);
    vector<pii> state(n, {INF, -1});
    vector<vector<int>> mst(n);
    state[start].i1 = 0;
    for (int i = 0; i < n; ++i) {
        int v = -1;
        for (int j = 0; j < n; ++j)
            if (!mark[j] && (v == -1 || state[j].i1 < state[v].i1))
                v = j;
        if (state[v].i1 == INF)
            return {};
        mark[v] = true;
        if (state[v].i2 != -1) {
            mst[v].push_back(state[v].i2);
            mst[state[v].i2].push_back(v);
        }
        for (int to = 0; to < n; ++to) {
            if (g[v][to] < state[to].i1) {
                state[to].i1 = g[v][to];
                state[to].i2 = v;
            }
        }
    }
    return mst;
}

// O(|E|log|E|) for sparse graph G=(V, E): |E| ~ |V|
vector<vector<int>> kruskal(vector<tuple<int, int, int>>& e_sorted, int n
     ) {
    vector<vector<int>> mst(n);
    dsu state(n);
    for (auto& [u, v, wt] : e_sorted)
        if (state.classify(u) != state.classify(v)) {
```

```cpp
            state.unite(u, v);
            mst[u].push_back(v);
            mst[v].push_back(u);
        }
    return mst;
}


// MATCHMAKING
// KUHN'S ALGORITHM FOR BIPARTITE GRAPHS O(|E|*|V|)
bool try_kuhn(vector<vector<int>>& g, vector<int>& match,
              vector<int>& mark, int v, int cur) {
    if (mark[v] == cur)
        return false;
    mark[v] = cur;
    for (auto u : g[v]) {
        if (match[u] == -1 || try_kuhn(g, match, mark, match[u], cur)) {
            match[u] = v;
            return true;
        }
    }
    return false;
}

vector<pii> kuhn(vector<vector<int>>& g, int l, int r) {
    int n = l+r;
    vector<int> match(n, -1);
    vector<int> greed(l, -1);
    for (int v = 0; v < l; ++v)
        for (int u = l; u < n; ++u)
            if (match[u] == -1) {
                match[u] = v;
                greed[v] = u;
                break;
            }
    vector<int> mark(n, 0);
    for (int v = 0; v < l; ++v)
        if (greed[v] == -1)
            try_kuhn(g, match, mark, v, v+1);
    vector<pii> res;
    for (int i = 0; i < r; ++i)
        if (match[l+i] != -1)
            res.emplace_back(match[l+i], l+i);
    return res;
}

// ********************************


// QUERIES ON TREE, LA & LCA
// *******************************

bool is_ancestor(const vector<int>& tin, const vector<int>& tout, int u,
     int v) {
    return tin[u] <= tin[v] && tin[v] < tout[u];
}

// BINUPS
// up[0..1][i] = 1 for each i

int time_counter = 0;

void build_binups(const vector<vector<int>>& g, vector<int>& tin, vector<
     int>& tout,
                  vector<vector<int>>& up, int logn, int u, int prev) {
    for (int i = 1; i < logn; ++i)
        up[u][i] = up[up[u][i-1]][i-1];
    tin[u] = time_counter++;
    for (auto& v : g[u]) {
        if (v == prev) continue;
        up[v][0] = u;
        build_binups(g, tin, tout, up, logn, v, u);
    }
    tout[u] = time_counter++;
}

int lca(const vector<vector<int>>& up, const vector<int>& tin, const
     vector<int>& tout,
       int logn, int u, int v) {
    if (is_ancestor(tin, tout, u, v)) return u;
    if (is_ancestor(tin, tout, v, u)) return v;
    for (int i = logn-1; i >= 0; --i)
        if (!is_ancestor(tin, tout, up[u][i], v))
            u = up[u][i];
    return up[u][0];
}


// TARJAN O(alpha(n)*(n+q))
void tarjan(vector<vector<int>>& g, vector<bool>& mark, dsu& state,
     vector<int>& anc,
            vector<vector<pii>>& q, vector<int>& res, int v) {
    mark[v] = true;
    anc[v] = v;

    for (auto u : g[v])
        if (!mark[u]) {
            tarjan(g, mark, state, anc, q, res, u);
            state.unite(v, u);
            anc[state.classify(v)] = v;
        }

    for (auto [u, pos] : q[v])
        if (mark[u])
            res[pos] = anc[state.classify(u)];
}


// CENTROID DECOMPOSITION
int dfs_sz(const vector<vector<int>>& g, vector<bool>& mark, vector<int>&
     sz, int v, int prev) {
    sz[v] = 1;
    for (auto& u : g[v]) {
        if (u == prev || mark[u]) continue;
        sz[v] += dfs_sz(g, mark, sz, u, v);
    }
    return sz[v];
}

int find_centroid(const vector<vector<int>>& g, const vector<bool>& mark,
     const vector<int>& sz,
                  int v, int prev, int n) {
    for (auto& u : g[v])
        if (u != prev && !mark[u] && 2*sz[u] > n)
            return find_centroid(g, mark, sz, u, v, n);
    return v;
}

void build_centroid_tree(const vector<vector<int>>& g, vector<bool>& mark
     , vector<int>& sz,
                         vector<int>& parcentr, int v, int c) {
    dfs_sz(g, mark, sz, v, v);
    int nc = find_centroid(g, mark, sz, v, v, sz[v]);
    parcentr[nc] = c;
    mark[nc] = 1;
    for (auto& u : g[nc]) {
        if (mark[u]) continue;
        build_centroid_tree(g, mark, sz, parcentr, u, nc);
    }
}


// HEAVY-LIGHT DECOMPOSITION
template<class T>
class hld {
public:
    struct node {
        vector<int> edg;
        int val;
    };

    hld(vector<node>& _g, function<int(int, int)> _op, int _defval);
    void dfs_sz(int v, int p);
    void build_hld(int v, int p);
    bool is_ancestor(int u, int v);
    void update(int v, int x);
    void up(int& u, int& v, int& res);
    int query(int u, int v);

    int n;
    vector<node> g;
    vector<int> sz, par, tin, tout, head;

    function<int(int,int)> op;
    int defval;
    T state;

private:
    int timer;
};

template<class T>
hld<T>::hld(vector<node>& _g, function<int(int, int)> _op, int _defval) :
       g(_g), op(_op),

       defval(_defval), state((int)g.size()) {
    n = (int)g.size();
    sz = par = tin = tout = head = vector<int>(n, 0);
    state = T(n);
    timer = 0;
}

template<class T>
void hld<T>::dfs_sz(int v, int p) {
    par[v] = p;
    sz[v] = 1;
    for (auto& u : g[v].edg) {
        if (u == p)
            continue;
        this->dfs_sz(u, v);
        sz[v] += sz[u];
        if (sz[u] > sz[g[v].edg[0]])
            swap(u, g[v].edg[0]);
    }
```

```cpp
}

template<class T>
void hld<T>::build_hld(int v, int p) {
    tin[v] = timer++;
    state.update(tin[v], g[v].val);
    for (auto u : g[v].edg) {
        if (u == p)
            continue;
        head[u] = (u == g[v].edg[0] ? head[v] : u);
        this->build_hld(u, v);
    }
    tout[v] = timer;
}

template<class T>
bool hld<T>::is_ancestor(int u, int v) {
    return tin[u] <= tin[v] && tin[v] < tout[u];
}

template<class T>
void hld<T>::update(int v, int x) {
    state.update(tin[v], x);
}

template<class T>
void hld<T>::up(int& u, int& v, int& res) {
    while (!this->is_ancestor(head[u], v)) {
        res = op(res, state.query(tin[head[u]], tin[u]));
        u = par[head[u]];
    }
}

template<class T>
int hld<T>::query(int u, int v) {
    int res = defval;
    this->up(u, v, res);
    this->up(v, u, res);
    if (!this->is_ancestor(u, v))
        swap(u, v);
    res = op(res, state.query(tin[u], tin[v]));
    return res;
}

// ********************************


// SPARSE TABLE
// ********************************

int rmq(int a, int b) {
    return max(a, b);
}

class sparse_table {
public:
    sparse_table(const vector<int>& source, function<int(int, int)>
     operation);
    ~sparse_table();
    int request(int i, int j);
private:
    int n;
    vector<int> logs;
    vector<vector<int>> st;
    function<int(int, int)> op;
};

sparse_table::sparse_table(const vector<int>& source, function<int(int,
     int)> operation) : op(operation) {
    n = source.size();
    logs = vector<int>(n+1, 0);
    logs[1] = 0;
    for (int i = 2; i <= n; ++i) logs[i] = logs[i/2]+1;
    int p = logs[n];
    st = vector<vector<int>>(n+1, vector<int>(p+1));
    for (int i = 0; i < n; ++i) st[i][0] = source[i];
    for (int j = 1; j <= p; ++j)
        for (int i = 0; i+(1<<j) <= n; ++i)
            st[i][j] = op(st[i][j-1], st[i+(1<<(j-1))][j-1]);
}

sparse_table::~sparse_table() {
    st.clear();
    logs.clear();
}

int sparse_table::request(int i, int j) {
    int k = logs[j-i+1];
    return op(st[i][k], st[j-(1<<k)+1][k]);
}

// ********************************


// SQRT DECOMPOSITION
// ********************************
```

```cpp
// ARRAY QUERIES (cin, cout instance)
void sqrt_queries() {
    int n;

    cin >> n;

    int s = sqrt(n);
    if (s*s < n) ++s;
    vector<int> a(s*s, 0);

    for (int i = 0; i < n; ++i) cin >> a[i];

    vector<int> pref(s, 0);
    for (int i = 0; i < n; ++i) pref[i/s] += a[i];
    int q;

    cin >> q;

    while (q--) {
        char t;
        cin >> t;
        if (t == '?') {
            int i, d;
            cin >> i >> d;
            --i;
            a[i] += d;
            pref[i/s] += d;

            cout << a[i] << endl;

        }
        else if (t == '+') {
            int u, v;
            cin >> u >> v;
            --u; --v;
            if (u/s == v/s) {
                int sum = 0;
                while (u <= v) {
                    sum += a[u];
                    ++u;
                }

                cout << sum << endl;

                continue;
            }
            int sum = 0;
            int i = u/s+1, j = v/s-1;
            for (int k = u; k/s < i; ++k) sum += a[k];
            for (int k = v; k/s > j; --k) sum += a[k];
            while (i <= j) {
                sum += pref[i];
                ++i;
            }

            cout << sum << endl;

        }
    }
}


// MO
struct query {
    int l, r, idx;
};
void mo(const vector<int>& a, const vector<vector<query>>& b, vector<int
     >& ans) {
    int n = (int)a.size(), c = (int)b.size(), q = (int)ans.size();
    for (int i = 0; i < c; ++i) {
        int l = i*c, r = i*c-1, res = 0;
        unordered_map<int, int> cnt;
        for (auto& q: b[i]) { // sorted
            while (r < q.r)
                if (cnt[a[++r]]++ == 0)
                    ++res;
            while (l < q.l)
                if (--cnt[a[l++]] == 0)
                    --res;
            while (l > q.l)
                if (cnt[a[--l]]++ == 0)
                    ++res;
            ans[q.idx] = res;
        }
    }
}
// cin, cout instance
void build_mo() {
    int n, q, c;
    vector<int> a, ans;
    vector<vector<query>> b;

    cin >> n >> q;

    c = (int)sqrt(n)+1;
    a.resize(n);
```

```cpp
    for (auto& item : a) cin >> item;

    ans.resize(q, 0);
    b.resize(c);
    for (int i = 0; i < q; ++i) {
        int l, r;

        cin >> l >> r;

        --l; --r;
        b[l/c].push_back({l, r, i});
    }
    for (auto& item : b)
        sort(item.begin(), item.end(), [](query& a, query& b) {
            return a.r < b.r;
        });
    mo(a, b, ans);

    for (auto& item : ans) cout << item << endl;

}

// ********************************


// SEGMENT TREE
// ********************************

// function "op" is any operation on MONOID
class point_segtree {
public:
    point_segtree(const vector<int>& a, function<int(int, int)> _op, int
      _defval);
    int at(int v);
    void update(int v, int l, int r, int idx, int val);
    int make_operation(int v, int tl, int tr, int l, int r);
private:
    void build(const vector<int>& a, int v, int l, int r);
    vector<int> tree;
    function<int(int, int)> op;
    int defval;
};

point_segtree::point_segtree(const vector<int>& a, function<int(int, int)
     > _op, int _defval)
        : op(_op), defval(_defval) {
    int n = a.size();
    tree.resize(4*n, defval);
    this->build(a, 0, 0, n);
}

void point_segtree::build(const vector<int>& a, int v, int l, int r) {
    if (r-l == 1) {
        tree[v] = a[l];
        return;
    }
    int mid = l + (r-l)/2;
    this->build(a, 2*v+1, l, mid);
    this->build(a, 2*v+2, mid, r);
    tree[v] = op(tree[2*v+1], tree[2*v+2]);
}

int point_segtree::at(int v) {
    return tree[v];
}

void point_segtree::update(int v, int l, int r, int idx, int val) {
    if (l > idx || r <= idx) return;
    if (r-l == 1) {
        tree[v] = val;
        return;
    }
    int mid = l + (r-l)/2;
    this->update(2*v+1, l, mid, idx, val);
    this->update(2*v+2, mid, r, idx, val);
    tree[v] = op(tree[2*v+1], tree[2*v+2]);
}

int point_segtree::make_operation(int v, int tl, int tr, int l, int r) {
    if (tl >= r || tr <= l) return defval;
    if (tl >= l && tr <= r) return tree[v];
    int mid = tl + (tr-tl)/2;
    return op(this->make_operation(2*v+1, tl, mid, l, r),
              this->make_operation(2*v+2, mid, tr, l, r));
}


// IMPLICIT SEGMENTS
// on intervals
class impl_point_segtree {
public:
    struct node {
        node(int _l, int _r, int _val) : val(_val), l(_l), r(_r) {}
        node* lc = nullptr;
        node* rc = nullptr;
        int l;
        int r;
        int val;
    };

    impl_point_segtree(int n, function<int(int, int)> _op, int _dv);
    void update(node* v, int i, int val);
    int query(node* v, int l, int r);

    node* root;
    function<int(int, int)> op;
    int defval;
};

impl_point_segtree::impl_point_segtree(int n, function<int(int, int)> _op
     , int _dv)
        : op(_op), defval(_dv) {
    root = new node(0, n, defval);
}

void impl_point_segtree::update(node* v, int i, int val) {
    if (v->l > i || v->r <= i)
        return;
    if (v->r - v->l == 1 && v->l == i) {
        v->val = val;
        return;
    }
    int mid = v->l + (v->r - v->l)/2;
    if (v->lc == nullptr)
        v->lc = new node(v->l, mid, defval);
    if (v->rc == nullptr)
        v->rc = new node(mid, v->r, defval);
    update(v->lc, i, val);
    update(v->rc, i, val);
    v->val = op(v->lc->val, v->rc->val);
}

int impl_point_segtree::query(node *v, int l, int r) {
    if (v->l >= r || v->r <= l)
        return defval;
    if (v->l >= l && v->r <= r)
        return v->val;
    if (v->lc == nullptr && v->rc == nullptr)
        return defval;
    if (v->lc == nullptr)
        return query(v->rc, l, r);
    if (v->rc == nullptr)
        return query(v->lc, l, r);
    return op(query(v->lc, l, r), query(v->rc, l, r));
}


// SEGMENT TREE (group update: =, function: stat (min/max))
// on intervals
class stat_segtree {
public:
    stat_segtree(const vector<int>& a, function<int(int, int)> _op, int
      _defval);
    int at(int v);
    void update(int v, int tl, int tr, int l, int r, int val);
    int query(int v, int tl, int tr, int l, int r);
private:
    void build(const vector<int>& a, int v, int l, int r);
    void make_push(int v);
    vector<int> tree;
    vector<int> push;
    function<int(int, int)> op;
    int defval;
};

stat_segtree::stat_segtree(const vector<int> &a, function<int(int, int)>
     _op, int _defval)
        : op(_op), defval(_defval) {
    int n = a.size();
    tree.resize(4*n, defval);
    push.resize(4*n, defval);
    this->build(a, 0, 0, n);
}

void stat_segtree::build(const vector<int> &a, int v, int l, int r) {
    if (r-l == 1) {
        tree[v] = a[l];
        return;
    }
    int mid = l + (r-l)/2;
    this->build(a, 2*v+1, l, mid);
    this->build(a, 2*v+2, mid, r);
    tree[v] = op(tree[2*v+1], tree[2*v+2]);
}

int stat_segtree::at(int v) {
    return tree[v];
}

void stat_segtree::update(int v, int tl, int tr, int l, int r, int val) {
    if (tl >= r || tr <= l) return;
    if (tl >= l && tr <= r) {
        push[v] = val;
        tree[v] = val;
```

```
        return;
    }
    make_push(v);
    int mid = tl + (tr-tl)/2;
    this->update(2*v+1, tl, mid, l, r, val);
    this->update(2*v+2, mid, tr, l, r, val);
    tree[v] = op(tree[2*v+1], tree[2*v+2]);
}

void stat_segtree::make_push(int v) {
    if (push[v] == defval || 2*v+2 >= (int)tree.size()) return;
    push[2*v+1] = push[v];
    push[2*v+2] = push[v];
    tree[2*v+1] = push[v];
    tree[2*v+2] = push[v];
    push[v] = defval;
}

int stat_segtree::query(int v, int tl, int tr, int l, int r) {
    if (tl >= r || tr <= l) return defval;
    if (tl >= l && tr <= r) return tree[v];
    make_push(v);
    int mid = tl + (tr-tl)/2;
    return op(this->query(2*v+1, tl, mid, l, r),
              this->query(2*v+2, mid, tr, l, r));
}


// SEGMENT TREE (group update: +=, function: sum or sum modulo)
// on segments
class cum_segment_tree {
public:
    cum_segment_tree(const vector<int>& a, function<int(int, int)> _op,
     int _defval);
    int at(int v);
    void update_segment(int v, int tl, int tr, int l, int r, int val);
    int make_operation(int v, int tl, int tr, int l, int r);
private:
    struct operation {
        int l, r, val;
    };
    void build(const vector<int>& a, int v, int l, int r);
    void make_push(int v);
    vector<int> tree;
    vector<operation> push;
    function<int(int, int)> op;
    int defval;
};

cum_segment_tree::cum_segment_tree(const vector<int> &a, function<int(int
     , int)> _op, int _defval)
        : op( _op), defval(_defval) {
    int n = a.size();
    tree.resize(4*n, defval);
    push.resize(4*n, {-1, -1, defval});
    this->build(a, 0, 0, n-1);
}

void cum_segment_tree::build(const vector<int> &a, int v, int l, int r) {
    push[v].l = l;
    push[v].r = r;
    if (l == r) {
        tree[v] = a[l];
        return;
    }
    int mid = (l+r)/2;
    this->build(a, 2*v+1, l, mid);
    this->build(a, 2*v+2, mid+1, r);
    tree[v] = op(tree[2*v+1], tree[2*v+2]);
}

int cum_segment_tree::at(int v) {
    return tree[v];
}

void cum_segment_tree::update_segment(int v, int tl, int tr, int l, int r
     , int val) {
    if (tl > r || tr < l) return;
    if (tl >= l && tr <= r) {
        push[v].val = op(push[v].val, val);
        tree[v] = op(tree[v], push[v].val * (push[v].r - push[v].l + 1));
        return;
    }
    make_push(v);
    int mid = (tl+tr)/2;
    this->update_segment(2*v+1, tl, mid, l, r, val);
    this->update_segment(2*v+2, mid+1, tr, l, r, val);
    tree[v] = op(tree[2*v+1], tree[2*v+2]);
}

void cum_segment_tree::make_push(int v) {
    if (push[v].val == defval || 2*v+2 >= (int)tree.size())
        return;
    push[2*v+1].val = op(push[2*v+1].val, push[v].val);
    push[2*v+2].val = op(push[2*v+2].val, push[v].val);
    tree[2*v+1] = op(tree[2*v+1], push[2*v+1].val * (push[2*v+1].r - push
     [2*v+1].l + 1));
    tree[2*v+2] = op(tree[2*v+2], push[2*v+2].val * (push[2*v+2].r - push
     [2*v+2].l + 1));
    push[v].val = defval;
}

int cum_segment_tree::make_operation(int v, int tl, int tr, int l, int r)
     {
    if (tl > r || tr < l) return defval;
    if (tl >= l && tr <= r) return tree[v];
    make_push(v);
    int mid = (tl+tr)/2;
    return op(this->make_operation(2*v+1, tl, mid, l, r),
              this->make_operation(2*v+2, mid+1, tr, l, r));
}


// MERGE SORT TREE: get first element x in [i; n]: x >= v
// on intervals
class merge_sort_tree {
public:
    merge_sort_tree(vector<int>& a);
    void upd(int v, int l, int r, int i, int x, int newx);
    void get_upbound(int v, int l, int r, int i, int x, int& res);
private:
    void build(vector<int>& a, int v, int l, int r);
    vector<multiset<int>> t;
};

merge_sort_tree::merge_sort_tree(vector<int>& a) {
    int n = (int)a.size();
    t = vector<multiset<int>>(4*n);
    build(a, 0, 0, n);
}

void merge_sort_tree::build(vector<int> &a, int v, int l, int r) {
    if (r-l == 1) {
        t[v] = { a[l] };
        return;
    }
    int m = l + (r-l)/2;
    build(a, 2*v+1, l, m);
    build(a, 2*v+2, m, r);
    t[v].insert(t[2*v+1].begin(), t[2*v+1].end());
    t[v].insert(t[2*v+2].begin(), t[2*v+2].end());
}

void merge_sort_tree::upd(int v, int l, int r, int i, int x, int newx) {
    if (l > i || r <= i)
        return;
    t[v].extract(x);
    t[v].insert(newx);
    if (r-l == 1)
        return;
    int m = l + (r-l)/2;
    upd(2*v+1, l, m, i, x, newx);
    upd(2*v+2, m, r, i, x, newx);
}

void merge_sort_tree::get_upbound(int v, int l, int r, int i, int x, int&
     res) {
    if (r <= i)
        return;
    if (l < i) {
        if (r-l == 1)
            return;
        int m = l + (r-l)/2;
        get_upbound(2*v+1, l, m, i, x, res);
        get_upbound(2*v+2, m, r, i, x, res);
        return;
    }
    auto found = t[v].upper_bound(x);
    if (found != t[v].end())
        res = min(res, *found);
}


// on intervals
// segment tree beats (a[i] = x, a[i..j] %= m, sum(l, r))
// potential   = SUM(V): SUM(X in segV): logX
class segment_tree_beats {
public:
    struct node { int l, r, sum, max; };
    segment_tree_beats(const vector<int>& a);
    void inc(int v, int i, int x);
    void mod(int v, int l, int r, int m);
    int sum(int v, int l, int r);
private:
    void build(const vector<int>& a, int v, int l, int r);
    vector<node> t;
};

segment_tree_beats::segment_tree_beats(const vector<int> &a) {
    int n = (int)a.size();
    t = vector<node>(4*n);
    build(a, 0, 0, n);
}
```

```
void segment_tree_beats::build(const vector<int> &a, int v, int l, int r)
    {
    t[v].l = l;
    t[v].r = r;
    if (r-l == 1) {
        t[v] = { l, r, a[l], a[l] };
        return;
    }
    int m = l + (r-l)/2;
    build(a, 2*v+1, l, m);
    build(a, 2*v+2, m, r);
    t[v].sum = t[2*v+1].sum + t[2*v+2].sum;
    t[v].max = max(t[2*v+1].max, t[2*v+2].max);
}

void segment_tree_beats::inc(int v, int i, int x) {
    if (t[v].l > i || t[v].r <= i)
        return;
    if (t[v].r - t[v].l == 1) {
        t[v].sum = x;
        t[v].max = x;
        return;
    }
    inc(2*v+1, i, x);
    inc(2*v+2, i, x);
    t[v].sum = t[2*v+1].sum + t[2*v+2].sum;
    t[v].max = max(t[2*v+1].max, t[2*v+2].max);
}

int segment_tree_beats::sum(int v, int l, int r) {
    if (t[v].l >= r || t[v].r <= l)
        return 0;
    if (t[v].l >= l && t[v].r <= r)
        return t[v].sum;
    return sum(2*v+1, l, r) + sum(2*v+2, l, r);
}

void segment_tree_beats::mod(int v, int l, int r, int m) {
    // break condition
    if (t[v].max < m || t[v].r <= l || t[v].l >= r)
        return;
    // tag condition
    if (t[v].r - t[v].l == 1) {
        t[v].sum %= m;
        t[v].max %= m;
        return;
    }
    mod(2*v+1, l, r, m);
    mod(2*v+2, l, r, m);
    t[v].sum = t[2*v+1].sum + t[2*v+2].sum;
    t[v].max = max(t[2*v+1].max, t[2*v+2].max);
}


// FENWICK
// a[1...n]
class fenwick {
public:
    fenwick(vector<int>* _a);
    void update(int i, int v);
    int sum(int l, int r);
private:
    int sum(int r);
    vector<int> t;
    vector<int>* a;
};

fenwick::fenwick(vector<int>* _a) {
    a = _a;
    t = vector<int>(a->size(), 0);
    for (int i = 1; i < t.size(); ++i)
        update(i, a->at(i));
}

int fenwick::sum(int r) {
    int res = 0;
    for (; r > 0; r -= r & -r)
        res += t[r];
    return res;
}

int fenwick::sum(int l, int r) {
    return sum(r) - sum(l-1);
}

void fenwick::update(int i, int v) {
    int delta = v - a->at(i);
    a->at(i) = v;
    for (; i < t.size(); i += i & -i)
        t[i] += delta;
}


// FENWICK WITH UPDATE (ADD) ON RANGE AND f IN POINT
class mass_fenwick {
public:
    mass_fenwick(int64_t n, int _n);
    void add(int l, int r, int v);
    int get(int i);
private:
    int n;
    fenwick t;
};

mass_fenwick::mass_fenwick(int64_t n, int _n) : n(_n), t(new vector<int>(
    _n, 0)) {}

void mass_fenwick::add(int l, int r, int v) {
    if (r < l || l <= 0)
        return;
    t.update(l, v);
    if (r < n)
        t.update(r+1, -v);
}

int mass_fenwick::get(int i) {
    return t.sum(1, i);
}

// ********************************



// CARTESIAN TREE
// ********************************

class treap { // random priority
public:
    struct node {
        int k, p, sz;
        node *l, *r;
        node(int _k);
    };
    treap();
    int size() const;
    const node* lower_bound(int k);
    const node* upper_rbound(int k);
    const node* upper_bound(int k);
    const node* at(int idx);
    const node* insert(int k);
    void erase(int k);
private:
    node* root;
    int eps; // minimal distance between keys ( = 1 in int keys )
    void update(node* v);
    node* merge(node* l, node* r);
    pair<node*, node*> split(node* v, int k);
    const node* lower_bound(const node* cur, int k);
    const node* upper_rbound(const node* cur, int k);
    const node* upper_bound(const node* cur, int k);
};

treap::node::node(int _k) {
    k = _k;
    sz = 1;
    p = global_rnd();
    l = r = nullptr;
}

treap::treap() {
    root = nullptr;
    eps = 1;
}

int treap::size() const {
    if (root == nullptr)
        return 0;
    return root->sz;
}


void treap::update(treap::node* v) {
    if (v == nullptr) return;
    v->sz = 1;
    if (v->l != nullptr) v->sz += v->l->sz;
    if (v->r != nullptr) v->sz += v->r->sz;
}

const treap::node* treap::at(int idx) {
    auto v = root;
    while (v != nullptr) {
        if ((v->l == nullptr && idx == 0) || (v->l != nullptr && v->l->sz
    == idx)) {
            return v;
        }
        if (v->l != nullptr && v->l->sz > idx) {
            v = v->l;
            continue;
        }
        if (v->r != nullptr && (v->l == nullptr || v->l->sz < idx)) {
            idx -= (v->l == nullptr ? 0 : v->l->sz) + 1;
            v = v->r;
            continue;
        }
```

```cpp
        return nullptr;
    }
    return v;
}

treap::node* treap::merge(node* l, node* r) {
    if (l == nullptr) return r;
    if (r == nullptr) return l;
    if (l->p <= r->p) {
        l->r = merge(l->r, r);
        update(l);
        return l;
    }
    r->l = merge(l, r->l);
    update(r);
    return r;
}

pair<treap::node*, treap::node*> treap::split(node* v, int k) {
    if (v == nullptr) return {v, v};
    if (v->k <= k) {
        auto [l, r] = split(v->r, k);
        v->r = l;
        update(v);
        return {v, r};
    }
    auto [l, r] = split(v->l, k);
    v->l = r;
    update(v);
    return {l, v};
}

const treap::node* treap::lower_bound(int k) {
    return this->lower_bound(root, k);
}

const treap::node* treap::lower_bound(const node* cur, int k) {
    if (cur == nullptr)
        return cur;
    if (k > cur->k)
        return lower_bound(cur->r, k);
    if (k == cur->k)
        return cur;
    auto v = lower_bound(cur->l, k);
    if (v != nullptr)
        return v;
    return cur;
}

const treap::node* treap::upper_rbound(int k) {
    return this->upper_rbound(root, k);
}

const treap::node* treap::upper_rbound(const node* cur, int k) {
    if (cur == nullptr)
        return cur;
    if (k <= cur->k)
        return upper_rbound(cur->l, k);
    auto v = upper_rbound(cur->r, k);
    if (v != nullptr)
        return v;
    return cur;
}

const treap::node* treap::upper_bound(int k) {
    return this->upper_bound(root, k);
}

const treap::node* treap::upper_bound(const node* cur, int k) {
    if (cur == nullptr)
        return cur;
    if (k >= cur->k)
        return upper_bound(cur->r, k);
    auto v = upper_bound(cur->l, k);
    if (v != nullptr)
        return v;
    return cur;
}

const treap::node* treap::insert(int k) {
    if (root == nullptr) {
        root = new node(k);
        return root;
    }
    auto v = lower_bound(root, k);
    if (v != nullptr && v->k == k)
        return v;
    auto n = new node(k);
    auto [l, r] = split(root, k);
    root = merge(merge(l, n), r);
}


void treap::erase(int k) {
    auto [l, cr] = split(root, k-eps);
    auto [c, r] = split(cr, k);
    root = merge(l, r);
    delete c;
}


// CARTESIAN TREE WITH IMPLICIT KEY (array with dynamic segments)
class itreap { // random priority
public:
    struct node {
        int sz, p, val;
        int dp; // result of operation on segment
        bool rev;
        node *l, *r;
        node(int val);
    };
    itreap(function<int(int, int)> _op, int _defval);
    int size() const;
    const node* at(int pos);
    vector<int> traverse(int l, int r);
    const node* insert(int pos, int val);
    void erase(int pos);
    void move(int l, int r, int pos);
    void reverse(int l, int r);
    int get_result(int l, int r);
private:
    function<int(int, int)> op;
    int defval;
    node* root;
    void update(node* v);
    node* merge(node* l, node* r);
    pair<node*, node*> split(node* v, int k);
    void traverse(vector<int>& a, node* where);
};

itreap::node::node(int _val) {
    sz = 1;
    p = global_rnd();
    val = _val;
    rev = false;
    l = r = nullptr;
}

itreap::itreap(function<int(int, int)> _op, int _defval) {
    op = _op;
    defval = _defval;
    root = nullptr;
}

void itreap::update(itreap::node* v) {
    if (v == nullptr)
        return;
    v->sz = 1;
    v->dp = v->val;
    if (v->rev)
        swap(v->l, v->r)
        ;
    if (v->l != nullptr) {
        v->sz += v->l->sz;
        v->dp = op(v->dp, v->l->dp);
        if (v->rev)
            v->l->rev ^= true;
    }
    if (v->r != nullptr) {
        v->sz += v->r->sz;
        v->dp = op(v->dp, v->r->dp);
        if (v->rev)
            v->r->rev ^= true;
    }
    v->rev = false;
}

const itreap::node* itreap::at(int pos) {
    auto [lc, r] = split(root, pos+1);
    auto [l, c] = split(lc, pos);
    auto res = c;
    root = merge(merge(l, c), r);
    return res;
}

vector<int> itreap::traverse(int i, int j) {
    auto [lc, r] = split(root, j+1);
    auto [l, c] = split(lc, i);
    vector<int> a;
    this->traverse(a, c);
    root = merge(merge(l, c), r);
    return a;
}

void itreap::traverse(vector<int> &a, node* where) {
    if (where == nullptr)
        return;
    update(where);
    if (where->l != nullptr)
        traverse(a, where->l);
    a.push_back(where->val);
    if (where->r != nullptr)
        traverse(a, where->r);
}
```

```
int itreap::size() const {
    if (root == nullptr)
        return 0;
    return root->sz;
}

itreap::node* itreap::merge(node* l, node* r) {
    if (l == nullptr) return r;
    if (r == nullptr) return l;
    update(l);
    update(r);
    if (l->p <= r->p) {
        l->r = merge(l->r, r);
        update(l);
        return l;
    }
    r->l = merge(l, r->l);
    update(r);
    return r;
}

pair<itreap::node*, itreap::node*> itreap::split(node* v, int k) {
    if (v == nullptr)
        return {v, v};
    update(v);
    int sz = v->l == nullptr ? 0 : v->l->sz;
    if (sz+1 <= k) {
        auto [l, r] = split(v->r, k-sz-1);
        v->r = l;
        update(v);
        return {v, r};
    }
    auto [l, r] = split(v->l, k);
    v->l = r;
    update(v);
    return {l, v};
}

const itreap::node* itreap::insert(int pos, int val) {
    auto n = new node(val);
    auto [l, r] = split(root, pos);
    root = merge(merge(l, n), r);
    return n;
}

void itreap::erase(int pos) {
    auto [lc, r] = split(root, pos+1);
    auto [l, c] = split(lc, pos);
    root = merge(l, r);
    delete c;
}

void itreap::move(int i, int j, int pos) {
    auto [lc, r] = split(root, j+1);
    auto [l, c] = split(lc, i);
    root = merge(l, r);
    auto [lp, rp] = split(root, pos);
    root = merge(merge(lp, c), rp);
}

void itreap::reverse(int i, int j) {
    auto [lc, r] = split(root, j+1);
    auto [l, c] = split(lc, i);
    if (c != nullptr)
        c->rev ^= true;
    root = merge(merge(l, c), r);
}

int itreap::get_result(int i, int j) {
    auto [lc, r] = split(root, j+1);
    auto [l, c] = split(lc, i);
    auto res = c == nullptr ? defval : c->dp;
    root = merge(merge(l, c), r);
    return res;
}

// ********************************



// GEOMETRY (on complex field)
// ********************************

float dot(cmpl v, cmpl u) {
    return (conj(v)*u).real();
}

float cross(cmpl v, cmpl u) {
    return (conj(v)*u).imag();
}


// CONVEX HULL
vector<cmpl> graham(vector<cmpl>& pts) { // points are unique
    if (pts.empty())
        return {};
    cmpl p0 = *pts.begin();
    for (auto& p : pts)
        if (p.real() < p0.real() || (p.real() == p0.real() && p.imag() <
 p0.imag()))
            p0 = p;
    pts.erase(find(pts.begin(), pts.end(), p0));
    sort(pts.begin(), pts.end(),
         [p0](cmpl a, cmpl b){
             auto prod = cross(a-p0, b-p0);
             if (prod > 0) return true;
             if (prod < 0) return false;
             return abs(a-p0) < abs(b-p0);
         }
    );
    vector<cmpl> hull = {p0};
    for (auto p : pts) {
        while (hull.size() >= 2) {
            cmpl f = p-hull.back(), s = hull.back()-hull[hull.size()-2];
            if (cross(f, s) > 0) hull.pop_back();
            else break;
        }
        hull.push_back(p);
    }
    return hull;
}

// ********************************
```