Caderno Zikados

Enrique Junchaya, Gustavo M. Carlos, Nathan Luiz Martins

config.md

~/.bashrc

```
setxkbmap -option caps:escape
```

~/.vimrc

```
set nu rnu sw=4 ts=4 ai ls=2
```

template.cpp

```
#include <bits/stdc++.h>
using namespace std;

#define fastio ios_base::sync_with_stdio(0);cin.tie(0)
#define pb push_back
#define mp make_pair
#define sz(x) int(x.size())
#define trace(x) cerr << #x << ": " << x << endl;

typedef long long ll;

const ll N = 1e6;
const ll INF = 1LL << 61;
const ll MOD = 1e9 + 7;

void solve() {

}

signed main() {
    fastio;
    solve();
    return 0;
}
```

Makefile

```
c = g++ -Wall -std=c++17 -static -lm $< -o $*
mtime = /usr/bin/time -f '%C %Us %MKB'

%: %.cpp
	$c -g

t%: %.cpp
	$c -O2
	@for i in $*.in*; do echo "\n== $$i ==" && $(mtime) ./$* < $$i; done
```

reference.md

bash

- Make CapsLock work as Esc
  `setxkbmap -option caps:escape`

- Remove gdb warranty message
  `alias gdb="gdb --silent"`
- Copy file to clipboard
  `xclip -sel c < file`
- Change keyboard language
  `setxkbmap us|br`

vim

commands

- Compile the program. If there are any errors, the cursor is moved to the first one
  `:make file`
- Go to the next error
  `:cnext`
- Show full error message
  `:cc`
- Show the list of errors
  `:clist`
- Split horizontaly
  `:split` `Ctrl-w s`
- Split verticaly
  `:vsplit` `Ctrl-w v`

options

```
" show line numbers (nu)
set number

" display relative line numbers instead of absolute (rnu)
set relativenumber

" the width of indent for operations like << and >> (sw)
set shiftwidth=4

" display size of the tab character (ts)
set tabstop=4

" use the indentation of the current line when creating a new one (ai)
set autoindent

" constantly show the file name (ls)
set laststatus=2

"" not used

" show folds in a column (fdc)
set foldcolumn=4

" fold by indent (fdm)
set foldmethod=indent

" set vim colorscheme (colo)
colorscheme delek

" autoindent when typing { and }
set smartindent

" number of spaces that a tab counts when typing <TAB> (sts)
set softtabstop=4

" inserted tabs turn into spaces (et)
set expandtab

" show information about the current position (activated by default, but just
" in case)
set ruler

" show tabs and trailing spaces
set list
```

Data structures

segment_tree_beats_1.cpp, iterative_segment_tree.cpp, persistent_segment_tree.cpp, iterative_lazy_segment_tree.cpp, iterative_segment_tree_2d.cpp, heavy_light_decomposition.cpp, ordered_set.cpp, median.cpp, search_buckets.cpp, sparse_table.cpp, fenwick_tree.cpp, implicit_lazy_treap.cpp, mos.cpp, monotonic_convex_hull_trick.cpp

segment_tree_beats_1.cpp

```
/* Level 1 of Segment tree beats. Can do range updates that assign to
 * every element in the range the minimum between its current value
 * and a given value x
 *
 * A - (input) initial array
 *
 * Complexity: O(logn) per query
 *             amortized O(logn) per update
 */

struct node{
    ll maxi_count, maxi, second, sum;
    bool leaf, lazy;
} t[N*4];

int A[N];

void build(int node) {
    t[node].maxi = max(t[node<<1].maxi, t[node<<1|1].maxi);
    t[node].sum = t[node<<1].sum + t[node<<1|1].sum;
    if(t[node<<1].maxi == t[node<<1|1].maxi) {
        t[node].maxi_count = t[node<<1].maxi_count + t[node<<1|1].maxi_count;
        t[node].second = max(t[node<<1].second, t[node<<1|1].second);
    } else if(t[node<<1].maxi > t[node<<1|1].maxi) {
        t[node].maxi_count = t[node<<1].maxi_count;
        t[node].second = max(t[node<<1|1].maxi, t[node<<1].second);
    } else {
        t[node].maxi_count = t[node<<1|1].maxi_count;
        t[node].second = max(t[node<<1].maxi, t[node<<1|1].second);
    }
}

void init(int l, int r, int node=1) {
    t[node].lazy = 0;
    t[node].leaf = 0;
    if(l == r) {
        t[node].maxi_count = 1;
        t[node].sum = t[node].maxi = A[l];
        t[node].second = -1;
        t[node].leaf = 1;
        return;
    }
    int m = (l + r) >> 1;
    init(l, m, node << 1);
    init(m + 1, r, node << 1 | 1);
    build(node);
}

void putTag(int node, ll x){
    t[node].lazy =  1;
    t[node].sum -= (t[node].maxi - x) * t[node].maxi_count;
    t[node].maxi = x;
}

void propagate(int node) {
    if(!t[node].leaf){
        if(t[node << 1].maxi > t[node].maxi)
            putTag(node << 1, t[node].maxi);
        if(t[node << 1 | 1].maxi > t[node].maxi)
            putTag(node <<1 | 1, t[node].maxi);
    }
    t[node].lazy = 0;
}

// Queries maximum in closed range [ll, rr]
int queMax(int ll, int rr, int l=0, int r=n-1, int node=1) {
    if(ll <= l && r <= rr) return t[node].maxi;
    if(rr < l || r < ll) return -1;
    if(t[node].lazy) propagate(node);
    int m = (l + r) >> 1;
    return max(queMax(ll, rr, l, m, node << 1), queMax(ll, rr, m+1, r, node << 1 | 1));
}


// Queries sum of closed range [ll, rr]
ll queSum(int ll, int rr, int l=0, int r=n-1, int node=1) {
    if(ll <= l && r <= rr) return t[node].sum;
    if(rr < l || r < ll) return 0;
    if(t[node].lazy) propagate(node);
    int m = (l + r) >> 1;
    return queSum(ll, rr, l, m, node << 1) + queSum(ll, rr, m + 1, r, node << 1 | 1);
}


// Updates to the minimum between x and current value in the closed range [ll, rr]
void upd(int ll, int rr, long long int x, int l=0, int r=n-1, int node=1) {
    if(rr < l || r < ll || t[node].maxi <= x)
        return;
    if(ll <= l && r <= rr && t[node].second < x) {
        putTag(node, x);
        return;
    }
    if(t[node].lazy) propagate(node);
    int m = (l + r) >> 1;
    upd(ll, rr, x, l, m, node << 1);
    upd(ll, rr, x, m+1, r, node << 1 | 1);
    build(node);
}
```

iterative_segment_tree.cpp

```
/* Simple iterative segment tree, with point update and range queries.
 *
 * The operator needs to be commutative for this implementation.
 *
 * Time complexity: O(n) for building and O(log n) for updates and queries.
 * Space complexity: O(n)
 */
struct segTree {
    int n;
    vector<ll> st;
    const ll NEUT = 0; // TODO define neutral element

    // combine two elements, doesn't need to be commutative
    inline ll combine(ll a, ll b) {
        return a + b; // TODO define merge operator
    }

    // build the tree with vector v
    void build(vector<ll> &v) {
        for (int i = 0; i < n; i++) {
            st[n + i] = v[i];
        }

        for (int i = n-1; i >= 1; i--) {
            st[i] = combine(st[i << 1], st[i << 1 | 1]);
        }
    }

    public:

    segTree() {}

    // initialize with neutral elements
    segTree(int n) {
        resize(n);
    }

    // initialize with vector
    segTree(vector<ll> &v) : segTree(v.size()) {
        build(v);
    }

    void resize(int s) {
        n = s;
        st.assign(2*s, NEUT);
    }

    // add x to position i
    void update(int i, ll x) {
        st[i += n] += x; // TODO change update operation
        while (i > 1) {
            i >>= 1;
            st[i] = combine(st[i << 1], st[i << 1 | 1]);
        }
    }

    // query from l to r, inclusive
    ll query(int l, int r) {
        ll resl = NEUT, resr = NEUT;
        for (l += n, r += n+1; l < r; l >>= 1, r >>= 1) {
            if (l & 1) resl = combine(resl, st[l++]);
            if (r & 1) resr = combine(st[--r], resr);
        }

        return combine(resl, resr);
    }
};
```

persistent_segment_tree.cpp

```
/* Persistent segment tree. This example is for queries of sum in range
 * and updates of sum in position, but any query or update can be achieved
 * changing the NEUT value, and functions updNode and merge.
 * The version 0 of the persistent segTree has implicitly an array of
 * length n full of NEUT values.
 *
 * It's recommend to set n as the actual length of the array.
 * ```
 * int n; cin >> n;
 * segTree::n = n;
 * ```
 *
 * Complexity: O(logn) memory and time per query/update
 */

const int NEUT = 0;

struct segTree {
    vector<int> t = vector<int>(1, NEUT);
    vector<int> left = vector<int>(1, 0), right = vector<int>(1, 0);
    static int n;
    int newNode(int v, int l=0, int r=0) {
        t.pb(v), left.pb(l), right.pb(r);
        return sz(t) - 1;
    }
    int merge(int a, int b) {
        return a + b;
    }
    // Initializes a segTree with the values of the array A of length n
    int init(int* A, int L=0, int R=n) {
        if(L + 1 == R) return newNode(A[L]);
        int M = (L + R) >> 1;
        int l = init(A, L, M), r = init(A, M, R);
        return newNode(merge(t[l], t[r]), l , r);
    }
    int updNode(int cur_value, int upd_value) {
        return cur_value + upd_value;
    }
    // updates the position pos of version k with the value v
    int upd(int k, int pos, int v, int L=0, int R=n) {
        int nxt = newNode(t[k], left[k], right[k]);
        if(L + 1 == R) t[nxt] = updNode(t[nxt], v);
        else {
            int M = (L + R) >> 1;
            int temp;
            if(pos < M) temp = upd(left[nxt], pos, v, L, M), left[nxt] = temp;
            else temp = upd(right[nxt], pos, v, M, R), right[nxt] = temp;
            t[nxt] = merge(t[left[nxt]], t[right[nxt]]);
        }
        return nxt;
    }
    // query in the range [l, r) of version k
    int que(int k, int l, int r, int L=0, int R=n){
        if(r <= L || R <= l) return NEUT;
        if(l <= L && R <= r) return t[k];
        int M = (L + R) >> 1;
        return merge(que(left[k], l, r, L, M), que(right[k], l, r, M, R));
    }
};
int segTree::n = N;
```

iterative_lazy_segment_tree.cpp

```
/* Iterative segment tree with lazy propagation. Supports range updates and queries.
 * This example is for querying the maximum value in a range and updating with sum in
 * a range.
 *
 * Changes must be done in the struct node operators nad NEUT value. For more complicated
 * lazy values, change the `apply` method in segTree too.
 *
 * Time complexity: O(n) for building and O(log n) for updates and queries.
 * Space complexity: O(n)
 */

const ll NEUT = -INF;

struct node {
    ll val;
    node() : val(0) {} // initial value
    node(ll val) : val(val) {}
    // combine two nodes
    node operator+(const node& other) {
        return node(max(val, other.val));
    }
    // update a node by the lazy value
    void operator+=(ll x) {
        val += x;
    }
};

struct segTree {
    int n, h;
    vector<ll> d;
    vector<node> t;
    segTree(int n) : n(n), t(n << 1) {
        d.resize(n, 0);
        h = sizeof(int) * 8 - __builtin_clz(n);
    }
    void apply(int p, ll x) {
        t[p] += x;
        if(p < n) d[p] += x;
    }
    void push(int p) {
        for(int s = h; s > 0; s--) {
            int i = p >> s;
            if(d[i]) {
                apply(i << 1, d[i]);
                apply(i << 1 | 1, d[i]);
                d[i] = 0;
            }
        }
    }
    void build(int p) {
        for(; p >>= 1;) {
            t[p] = t[p << 1] + t[p << 1 | 1];
            t[p] += d[p];
        }
    }
    void update(int l, int r, ll x){
        l += n, r += n;
        int l0 = l, r0 = r;
        push(l);
        push(r - 1);
        for(; l < r; l >>= 1, r >>= 1) {
            if(l & 1) apply(l++, x);
            if(r & 1) apply(--r, x);
        }
        build(l0);
        build(r0 - 1);
    }
    node query(int l, int r) {
        l += n, r += n;
        push(l);
        push(r - 1);
        node ans(NEUT);
        for(; l < r; l >>= 1, r >>= 1){
            if(l & 1) ans = ans + t[l++];
            if(r & 1) ans = ans + t[--r];
        }
        return ans;
    }
};
```

iterative_segment_tree_2d.cpp

```
/* 2d iterative segment tree, with point update and range queries.
 *
 * The operator needs to be commutative for this implementation.
 *
 * From: https://github.com/mhunicken/icpc-team-notebook-el-
vasito/blob/master/data_structures/segment_tree_2d.cpp
 *
 * Time complexity: O(n*m) for building and O(log n * log m) for updates and queries.
 * Space complexity: O(n*m)
 */
struct segTree {
    int n, m;
    vector<vector<ll>> st;
    const ll NEUT = 0; // TODO define neutral element

    // combine two elements, needs to be commutative
    inline ll combine(ll a, ll b) {
        return a + b; // TODO define merge operator
    }

    // build the tree with matriz mat
    void build(vector<vector<ll>> &mat) {
        for (int i = 0; i < n; i++) for (int j = 0; j < m; j++)
            st[n + i][m + j] = mat[i][j];

        for (int i = 0; i < n; i++) for (int j = m-1; j >= 1; j--)
            st[n + i][j] = combine(st[n + i][j << 1], st[n + i][j << 1 | 1]);

        for (int i = n-1; i >= 1; i--) for (int j = 0; j < 2*m; j++)
            st[i][j] = combine(st[i << 1][j], st[i << 1 | 1][j]);
    }

    public:

    segTree() {}

    // initialize with neutral elements
    segTree(int n, int m) {
        resize(n, m);
    }

    // initialize with matrix
    segTree(vector<vector<ll>> &m) : segTree(m.size(), m.front().size()) {
        build(m);
    }

    void resize(int new_n, int new_m) {
        n = new_n;
        m = new_m;
        st.assign(2*n, vector<ll>(2*m, NEUT));
    }

    // set position (x, y) to k
    void update(int x, int y, ll k) {
        st[n + x][m + y] = k; // TODO change update operation

        for (int j = m + y; j > 1; j >>= 1)
            st[n + x][j >> 1] = combine(st[n + x][j], st[n + x][j ^ 1]);

        for (int i = n + x; i > 1; i >>= 1) for (int j = m + y; j >= 1; j >>= 1)
            st[i >> 1][j] = combine(st[i][j], st[i ^ 1][j]);
    }

    // query in the rectangle (is, js) (ie, je), INCLUSIVE !!!
    ll query(int is, int js, int ie, int je) {
        ll res = NEUT;

        for (int i0 = n + is, i1 = n + ie + 1; i0 < i1; i0 >>= 1, i1 >>= 1) {
            ll t[2], q = 0;
            if (i0 & 1) t[q++] = i0++;
            if (i1 & 1) t[q++] = --i1;
            for (int k = 0; k < q; k++) {
                for (int j0 = m + js, j1 = m + je + 1; j0 < j1; j0 >>= 1, j1 >>= 1) {
                    if (j0 & 1) res = combine(res, st[t[k]][j0++]);
                    if (j1 & 1) res = combine(res, st[t[k]][--j1]);
                }
            }
        }

        return res;
    }
};
```

heavy_light_decomposition.cpp

```
/* Data structure to answer queries on paths of a tree. It divides the tree in chains
 * and for each chain saves a data structure that answers the query as if it were done
 * in an array. In this template, this data structure is assumed to be a segment tree
 * but that's not mandatory.
 *
 * Complexity: O(T(n) * logn) time per query/update, where T(n) is the time of
 *             query/update of the inherent data structure.
 */

struct hld {
    int gid, r;
    vector<int> tam, id, p, d, rt;
    vector<vector<int> > adj;
    segTree st;
    hld(int n, int root=0)
       : r(root), gid(0), tam(n), id(n), p(n), d(n), rt(n), adj(n), st(n) {}
    void addEdge(int u, int v) {
        adj[u].pb(v);
        adj[v].pb(u);
    }
    int prec(int v, int par=-1, int depth=0){
        tam[v] = 1;
        p[v] = par;
        d[v] = depth;
        for(int u: adj[v]) if(u != par)
            tam[v] += prec(u, v, depth + 1);
        return tam[v];
    }
    void build(int v, int root){
        id[v] = gid++;
        rt[v] = root;
        if(sz(adj[v]) > 1 && adj[v][0] == p[v])
            swap(adj[v][0], adj[v][1]);
        for(auto &u: adj[v]) if(u != p[v] && tam[u] > tam[adj[v][0]])
            swap(adj[v][0], u);
        for(auto u: adj[v]) if(u != p[v])
            build(u, u == adj[v][0] ? root : u);
    }
    void init(){
        prec(r);
        build(r, r);
    }
    void updateVertex(int u, int x){
        st.update(id[u], x);
    }
    void updateEdge(int u, int v, int x) {
        st.update(max(id[u], id[v]), x);
    }
    // for queries on the edges of the path set for_edge to true
    // this code assumes that the segment tree queries are right-exclusive
    int query(int u, int v, bool for_edge=false){
        auto oper = (ll a, ll b) { // you probably will only need to change this
            return max(a, b);
        };
        ll ans = -INF;
        while(rt[u] != rt[v]){
            if(d[rt[u]] > d[rt[v]]) swap(u, v);
            ans = oper(ans, st.query(id[rt[v]], id[v] + 1));
            v = p[rt[v]];
        }
        int a = id[u], b = id[v];
        ans = oper(ans, st.query(min(a, b) + for_edge, max(a, b) + 1));
        return ans;
    }
};
```

ordered_set.cpp

```
/* The ordered set data structure works just like the std::set, but it has additional
 * information for every node. Namely, it can retrieve the order for a given key and also
 * the key of a node which is in a given position.
 *
 * Methods:
```

```
 * - find_by_order(pos)
 *   returns the iterator of the node in the `pos` position.
 * - order_of_key(key)
 *   returns the order in which a node of key `key` would be in the ordered set
 * The other methods are the same of set's methods.
 *
 * Time complexity: O(logn) for insert, erase and queries
 * Space complexity: O(n)
 */

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

template<typename T>
using ordered_set = tree<T, null_type, less<T>, rb_tree_tag, tree_order_statistics_node_update>;
```

median.cpp

```
struct Median {
    multiset<ll, greater<ll>> sm;
    multiset<ll> gt;
    int size = 0;

    void balance() {
        if (sm.size() < (size + 1) / 2) {
            ll x = *(gt.begin());
            gt.erase(gt.begin());
            sm.insert(x);
        } else if (sm.size() > (size + 1) / 2) {
            ll x = *(sm.begin());
            sm.erase(sm.begin());
            gt.insert(x);
        }
    }

    void insert(ll v) {
        if (size == 0 || v <= median_first()) sm.insert(v);
        else gt.insert(v);
        size++;
        balance();
    }

    bool remove(ll v) {
        if (size == 0) return false;
        if (v <= median_first()) {
            auto it = sm.find(v);
            if (it == sm.end()) return false;
            sm.erase(it);
        } else {
            auto it = gt.find(v);
            if (it == gt.end()) return false;
            gt.erase(it);
        }
        size--;
        balance();
        return true;
    }

    ll median_first() {
        if (size % 2 == 1) return *(sm.begin());
        return *(sm.begin());
    }

    double median() {
        if (size % 2 == 1) return *(sm.begin());
        return (*(sm.begin()) + *(gt.begin())) / 2.0;
    }

    string median_string() {
        if (size % 2 == 1) return to_string(*(sm.begin()));
        ll a = *(sm.begin()) + *(gt.begin());
        string s;
        if (a < 0) s += '-';
        s += to_string(abs(a) / 2);
        if (abs(a) % 2 == 1) s += ".5";
        return s;
    }
};
```

search_buckets.cpp

```
#include <algorithm>
#include <cassert>
#include <cmath>
#include <iostream>
#include <vector>
using namespace std;


// search_buckets provides two operations on an array:
// 1) set array[i] = x
// 2) count how many i in [start, end) satisfy array[i] < value
// Both operations take sqrt(N log N) time. Amazingly, because of the cache efficiency this is
faster than the
// (log N)^2 algorithm until N = 2-5 million.
template<typename T>
struct search_buckets {
    // values are just the values in order. buckets are sorted in segments of BUCKET_SIZE (last
segment may be smaller)
    int N, BUCKET_SIZE;
    vector<T> values, buckets;

    search_buckets(const vector<T> &initial = {}) {
        init(initial);
    }

    int get_bucket_end(int bucket_start) const {
        return min(bucket_start + BUCKET_SIZE, N);
    }

    void init(const vector<T> &initial) {
        values = buckets = initial;
        N = values.size();
        BUCKET_SIZE = 3 * sqrt(N * log(N + 1)) + 1;
        cerr << "Bucket size: " << BUCKET_SIZE << endl;

        for (int start = 0; start < N; start += BUCKET_SIZE)
            sort(buckets.begin() + start, buckets.begin() + get_bucket_end(start));
    }

    int bucket_less_than(int bucket_start, T value) const {
        auto begin = buckets.begin() + bucket_start;
        auto end = buckets.begin() + get_bucket_end(bucket_start);
        return lower_bound(begin, end, value) - begin;
    }

    int less_than(int start, int end, T value) const {
        int count = 0;
        int bucket_start = start - start % BUCKET_SIZE;
        int bucket_end = min(get_bucket_end(bucket_start), end);

        if (start - bucket_start < bucket_end - start) {
            while (start > bucket_start)
                count -= values[--start] < value;
        } else {
            while (start < bucket_end)
                count += values[start++] < value;
        }

        if (start == end)
            return count;

        bucket_start = end - end % BUCKET_SIZE;
        bucket_end = get_bucket_end(bucket_start);

        if (end - bucket_start < bucket_end - end) {
            while (end > bucket_start)
                count += values[--end] < value;
        } else {
            while (end < bucket_end)
                count -= values[end++] < value;
        }

        while (start < end && get_bucket_end(start) <= end) {
            count += bucket_less_than(start, value);
            start = get_bucket_end(start);
        }

        assert(start == end);
        return count;
    }
    int prefix_less_than(int n, T value) const {
        return less_than(0, n, value);
    }

    void modify(int index, T value) {
        int bucket_start = index - index % BUCKET_SIZE;
        int old_pos = bucket_start + bucket_less_than(bucket_start, values[index]);
        int new_pos = bucket_start + bucket_less_than(bucket_start, value);

        if (old_pos < new_pos) {
            copy(buckets.begin() + old_pos + 1, buckets.begin() + new_pos, buckets.begin() +
old_pos);
            new_pos--;
            // memmove(&buckets[old_pos], &buckets[old_pos + 1], (new_pos - old_pos) * sizeof(T));
        } else {
            copy_backward(buckets.begin() + new_pos, buckets.begin() + old_pos, buckets.begin() +
old_pos + 1);
            // memmove(&buckets[new_pos + 1], &buckets[new_pos], (old_pos - new_pos) * sizeof(T));
        }

        buckets[new_pos] = value;
        values[index] = value;
    }
};


int main() {
    int N, M;
    scanf("%d %d", &N, &M);
    vector<int> A(N), B(N);
    vector<int> location(N + 1);

    for (int i = 0; i < N; i++) {
        scanf("%d", &A[i]);
        location[A[i]] = i;
    }

    for (int &b : B) {
        scanf("%d", &b);
        b = location[b];
    }

    search_buckets<int> buckets(B);

    for (int i = 0; i < M; i++) {
        int type;
        scanf("%d", &type);

        if (type == 1) {
            int LA, RA, LB, RB;
            scanf("%d %d %d %d", &LA, &RA, &LB, &RB);
            LA--; LB--;
            printf("%d\n", buckets.less_than(LB, RB, RA) - buckets.less_than(LB, RB, LA));
        } else if (type == 2) {
            int x, y;
            scanf("%d %d", &x, &y);
            x--; y--;
            buckets.modify(x, B[y]);
            buckets.modify(y, B[x]);
            swap(B[x], B[y]);
        } else {
            assert(false);
        }
    }
}
```

sparse_table.cpp

```
/* Sparse table. Useful for queries of idempotent functions in a range.
 * Change the oper method accordingly.
 *
 * Time complexity:
 *   - O(n logn) for building the structure
 *   - O(1) for queries of idempotent functions.
 * Space complexity: O(n logn)
 */

struct sparseTable {
    int n, logn;
    vector<vector<ll>> t;
    int log_floor(int tam) {
        return 31 - __builtin_clz(tam);
    }
    ll oper(ll a, ll b) { // example with minimum in range
        return min(a, b);
    }
    sparseTable() {}
    sparseTable(vector<ll>& v) {
        n = sz(v);
        logn = log_floor(n) + 1;
        t.resize(logn);
        t[0].resize(n);
        for(int i = 0; i < n; i++)
            t[0][i] = v[i];
        for(int k = 1; k < logn; k++) {
            t[k].resize(n);
            for(int i = 0; i + (1 << k) <= n; i++)
                t[k][i] = oper(t[k - 1][i], t[k - 1][i + (1 << (k - 1))]);
        }
    }
    ll que(int l, int r) { // queries in the semi-open interval [l, r)
        int k = log_floor(r - l);
        return oper(t[k][l], t[k][r - (1 << k)]);
    }
};
```

fenwick_tree.cpp

```
/* Fenwick tree, with point update and range queries.
 *
 * The operator needs to be commutative for this implementation.
 *
 * Time complexity: O(n log(n)) for building and O(log n) for updates and queries.
 * Space complexity: O(n)
 */
struct Fenwick {
    vector<ll> bit;
    int n;

    void build(vector<ll> &v) {
        for (size_t i = 0; i < v.size(); i++) {
            update(i, v[i]);
        }
    }

    Fenwick() {}

    Fenwick(int n) : n(n) {
        resize(n);
    }

    Fenwick(vector<ll> &v) : Fenwick(v.size()) {
        build(v);
    }

    void resize(int n) {
        bit.assign(n, 0);
    }

    ll query(int r) {
        ll ret = 0;
        for (; r >= 0; r = (r & (r + 1)) - 1) {
            ret += bit[r];
        }
        return ret;
    }

    ll query(int l, int r) {
        return query(r) - query(l-1);
    }

    void update(int i, ll v) {
        for (; i < n; i = i | (i + 1)) {
            bit[i] += v;
        }
    }
};
```

implicit_lazy_treap.cpp

```
/* Implicit treap with lazy propagation. It is easy to adapt to versions without
 * lazy propagation or that are not implicit.
 * For non-lazy versions, just delete the functions apply and push.
 * For non-implicit versions, use an actual key in the split instead of the implicit key.
 *
 * This example uses lazy propagation for reversing a range in implicit treaps.
 *
 * Time complexity: O(log n) for merge and split.
 * Space complexity: O(n)
 */

mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());

typedef struct item* pitem;
struct item {
    int cnt, pr, val;
    bool lazy;
    pitem l, r;
    item(int val)
        : pr(rng()), val(val), cnt(1), lazy(0), l(nullptr), r(nullptr) {}
};

int cnt(pitem t) {
    return t != nullptr ? t->cnt : 0;
}

void upd(pitem t) {
    if(t != nullptr)
        t->cnt = cnt(t->l) + 1 + cnt(t->r);
}

void apply(pitem t, bool d) {
    if(t != nullptr) {
        if(d.first) swap(t->l, t->r);
        t->lazy ^= d;
    }
}

void push(pitem t) {
    if(t != nullptr) {
        apply(t->l, t->lazy);
        apply(t->r, t->lazy);
        t->lazy = {0, 0};
    }
}

void merge(pitem& t, pitem l, pitem r) {
    if(l == nullptr || r == nullptr)
        t = l != nullptr ? l : r;
    else if(l->pr > r->pr) {
        push(l);
        merge(l->r, l->r, r), t = l;
    } else {
        push(r);
        merge(r->l, l, r->l), t = r;
    }
    upd(t);
}

void split(pitem t, pitem& l, pitem& r, int key, int add=0) {
    if(t == nullptr) {
        l = r = nullptr; return;
    }
    push(t);
    int cur_key = add + cnt(t->l);
    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(t);
}

void insert(pitem& t, int pos) {
    pitem it = new item(pos);
    pitem t1, t2;
    split(t, t1, t2, pos);
    merge(t1, t1, it);
    merge(t, t1, t2);
}
```

mos.cpp

```
/* Optimization for problems with queries that can be answered offline
 *
 * Complexity: O(q * (BLOCK_SIZE) + n * (n / BLOCK_SIZE))
 *             O((q + n) * sqrt(n)) if BLOCK_SIZE ~ sqrt(n)
 */

const int BLOCK_SIZE = 700;

struct Query {
    int l, r, idx;
    bool operator<(Query other) const
    {
        return make_pair(l / BLOCK_SIZE, r) <
               make_pair(other.l / BLOCK_SIZE, other.r);
    }
};

struct Mos {
    // variables specific for the problem
    ll sum;
    vector<ll> v;

    // stores the answers to the queries in the original order
    void exec(vector<Query> &queries, vector<ll> &answers) {
        answers.resize(queries.size());
        sort(queries.begin(), queries.end());

        int cur_l = 0;
        int cur_r = -1;

        for (Query q : queries) {
            while (cur_l > q.l) {
                cur_l--;
                add(cur_l);
            }
            while (cur_r < q.r) {
                cur_r++;
                add(cur_r);
            }
            while (cur_l < q.l) {
                remove(cur_l);
                cur_l++;
            }
            while (cur_r > q.r) {
                remove(cur_r);
                cur_r--;
            }
            answers[q.idx] = get_answer(cur_l, cur_r);
        }
    }

    // functions below are specific for the problem
    Mos(vector<ll> &v) : sum(0), v(v) {}

    void add(int i) {
        sum += v[i];
    }

    void remove(int i) {
        sum -= v[i];
    }

    ll get_answer(int l, int r) {
        return sum;
    }
};
```

monotonic_convex_hull_trick.cpp

```
/* Convex hull trick version for monotonic slopes and queries, this is,
 * the inserted lines have only increasing/decreasing slopes and the
 * queries are done only in increasing/decreasing x-coordinates.
 *
 * This template assumes non-decreasing slopes in inserted lines
 * and queries of maximization in non-decreasing x-coordinates.
 *
 * Complexity: amortized O(1) for each insertion and query
 */

typedef long double ld;
```

```
struct chullTrick {
    deque<pair<ll, ll> > lines;
    ll eval(ll x, pair<ll, ll> line) {
        return line.first*x + line.second;
    }
    ld inter(pair<ll, ll> a, pair<ll, ll> b) {
        return ld(b.second - a.second) / (a.first - b.first);
    }
    ll que(ll x) {
        while(sz(lines) >= 2 && eval(x,lines[0]) <= eval(x,lines[1]))
            lines.pop_front();
        return eval(x, lines[0]);
    }
    void insert(pair<ll, ll> nline) {
        while(sz(lines) >= 2 && inter(nline,lines[sz(lines)-2]) <
inter(lines.back(),lines[sz(lines)-2]) + EPS)
            lines.pop_back();
        if(sz(lines) == 1 && lines.back().first == nline.first && lines.back().second <
nline.second)
            lines.pop_back();
        if(lines.empty() || nline.first != lines.back().first)
            lines.push_back(nline);
    }
};
```

Dynamic programming

longest_increasing_subsequence.cpp, knuth.cpp, divide_and_conquer.cpp

longest_increasing_subsequence.cpp

```
/* Computes the longest (weakly) increasing subsequence in a vector.
 * Is easy to change the code below for the strictly increasing version of the problem.
 *
 * v - (input) vector for which the longest increasing subsequence will be computed.
 * aux - one of the longest increasing subsequences of v
 *
 * Complexity: O(n logn) time
 *             O(n) memory
 */

vector<int> lis(vector<int>& v) {
    vector<int> aux;
    for(auto x: v) {
        // Change to lower_bound for strictly increasing version
        auto it = upper_bound(aux.begin(), aux.end(), x);
        if(it == aux.end()) aux.pb(x);
        else *it = x;
    }
    return aux;
}
```

knuth.cpp

```
/* A dp of the form
 *     dp[l][r] = min_{l < m < r}(dp[l][m] + dp[m][r])  + cost(l, r)
 * can be solved in O(n^2) with Knuth opmitization if we have that
 *     opt[l][r - 1] <= opt[l][r] <= opt[l + 1][r]
 * where
 *     dp[l][r] = dp[l][opt[l][r]] + dp[opt[l][r]][r] + cost(l, r).
 *
 * Other sufficient condition (that implies the previous one) is
 *    given a <= b <= c <= d, we have:
 *        - quadrangle inequality: cost(a, c) + cost(b, d) <= cost(a, d) + cost(b, c)
 *        - monotonicity: cost(b, c) <= cost(a, d)
 *
 * Complexity: O(n^2) time
 *             O(n^2) memory
 */

ll knuth(int n) {
    vector<vector<ll>> dp = vector<vector<ll>>(n, vector<ll>(n));
    vector<vector<int>> opt = vector<vector<int>>(n, vector<int>(n));
    for(int k = 0; k <= n; k++) {
        for(int l = 0; l + k <= n; l++) {
            int r = l + k;
            if(k < 2) {
                dp[l][r] = 0; // base case
                opt[l][r] = l;
            }
            dp[l][r] = INF;
            for(int m = opt[l][r - 1]; m <= opt[l + 1][r]; m++) {
                ll cur = dp[l][m] + dp[m][r] + cost(l, r); // must define O(1) cost function
                if(cur < dp[l][r]) {
                    dp[l][r] = cur;
                    opt[l][r] = m;
                }
            }
        }
    }
    return dp[0][n];
}
```

divide_and_conquer.cpp

```
/* A dp of the form
 *     dp[i][j] = min_{k < j}(dp[i - 1][k]  + cost(k, j))
 * can be solved in O(m n logn) with divide and conquer optimization if we have that
 *     opt[i][j] <= opt[i][j + 1]
 * where
 *     dp[i][j] = dp[i - 1][opt[i][j]]  + cost(opt[i][j], j).
 *
 * Complexity: O(m n logn) time (for a partition in m subarrays of an array of size n)
 *             O(n) memory
 */

ll dp[N][2];

void go(int k, int l, int r, int optl, int optr) {
    if(l > r) return;
    int opt, m = (l + r) >> 1;
    dp[m][k&1] = INF;
    for(int i = optl; i <= min(m, optr + 1); i++) {
        ll cur = dp[i][~k&1] + cost(i, m); // must define O(1) cost function
        if(cur < dp[m][k&1]) {
            dp[m][k&1] = cur;
            opt = i;
        }
    }
    go(k, l, m - 1, optl, opt);
    go(k, m + 1, r, opt, optr);
}

ll dc(int n, int m) {
    dp[0][0] = dp[0][1] = 0;
    for(int i = 0; i < n; i++) dp[i][0] = INF;
    for(int i = 1; i <= m; i++) go(i, i , n, i - 1, n - 1);
    return dp[n - 1][m & 1];
}
```

Flows and matchings

bipartite_matching.cpp, dinic.cpp, hungarian.cpp, stable_matching.cpp, flow_with_demands.cpp, min_cost_max_flow_dijkstra.cpp, min_cost_max_flow_spfa.cpp

bipartite_matching.cpp

```
/* Solves the bipartite matching problem for color classes X and Y.
 *
 * n - (input) size of color class X
 * m - (input) size of color class Y
 * g - (input) g[x] contains the neighbours of x \in X
 * mat - mat[y] is the vertex matched with y \in Y or -1 if there's no such vertex
 * arcs - set of arcs of the bipartite matching. Each arc is a pair such that the first
          element is in X and the second in Y.
 *
 * Complexity: O(V * (V + E)) where V is the number of vertex and E the number of edges
                              in the bipartite graph
 */

vector<int> g[N];
int mat[N];
bool vis[N];
int n, m;

int match(int x) {
    if(vis[x]) return 0;
    vis[x] = true;
    for(int y: g[x]) if(mat[y] < 0 || match(mat[y])) {
        mat[y] = x;
        return 1;
    }
    return 0;
}

vector<pair<int, int> > maxMatching() {
    vector<pair<int, int> > arcs;
    for(int i = 0; i < m; i++)
        mat[i] = -1;
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++)
            vis[j] = 0;
        match(i);
    }
    for(int i = 0; i < m; i++)
        if(mat[i] >= 0) arcs.pb({mat[i], i});
    return arcs;
}
```

dinic.cpp

```
/* Fast max flow algorithm.
 *
 * Constructor:
 * dinic(n, s, t)
 * n - number of nodes in the flow network.
 * s - source of the flow network.
 * t - sink of the flow network.
 *
 * Methods:
 * - addEdge(u, v, cap)
 *   adds a directed edge from `u` to `v` with capacity `cap`.
 * - getFlow()
 *   returns the maximum flow of the network.
 *
 * Complexity: In general, the time complexity of getFlow is O(E V^2), but there are better
 *             upper bounds for bipartite graphs (O(E sqrt(V))) and networks with unit
 *             capacities (O(E sqrt(E))).
 */

const int INF = 1 << 29;

struct dinic {
    ll n, s, t;
    vector<ll> dist, q, work;
    struct edge {
        ll to, rev, f, cap;
    };
    vector<vector<edge> > g;
    dinic(int n, int s, int t) : n(n), s(s), t(t), g(n), dist(n), q(n), work(n) {}
    void addEdge(int u, int v, ll cap) {
        g[u].pb((edge){v, sz(g[v]), 0, cap});
        g[v].pb((edge){u, sz(g[u]) - 1, 0, 0});
    }
    bool bfs() {
        for(int i = 0; i < n; i++) dist[i] = -1;
        dist[s] = 0;
        int qt = 0;
        q[qt++] = s;
        for(int qh = 0; qh < qt; qh++) {
            int u = q[qh];
            for(int i = 0; i < sz(g[u]); i++) {
                edge &e = g[u][i];
                int v = g[u][i].to;
                if(dist[v] < 0 && e.f < e.cap) {
                    dist[v] = dist[u] + 1;
                    q[qt++]=v;
                }
            }
        }
        return dist[t] >= 0;
    }
    ll dfs(int u, ll f) {
        if(u == t) return f;
        for(ll &i = work[u]; i < sz(g[u]); i++) {
            edge &e = g[u][i];
            if(e.cap <= e.f) continue;
            int v = e.to;
            if(dist[v] == dist[u] + 1) {
                ll df = dfs(v, min(f, e.cap - e.f));
                if(df > 0){
                    e.f += df;
                    g[v][e.rev].f -= df;
                    return df;
                }
            }
        }
        return 0;
    }
    ll getFlow() {
        ll res = 0;
        while(bfs()) {
            for(int i = 0; i < n; i++) work[i] = 0;
            while(ll delta = dfs(s, INF))
                res += delta;
        }
        return res;
    }
};
```

hungarian.cpp

```
/* Hungarian method algorithm that solves the bipartite perfect matching of minimum cost.
 * Can solve the problem of maximum cost with minor changes.
 *
 * Constructor:
 * hungarian(n, m)
 * n - (input) size of color class X
 * m - (input) size of color class Y
 *
 * Methods:
 * - set(x, y, c)
 *   sets the cost c for the edge between x \in X to y \in Y
 * - assign()
 *   returns the cost of an optimal matching and fills the vectors matchl and matchr
 *   with the assignment of such matching
 * Complexity: O(V^3) where V is the number of vertex of the graph
 */

typedef long double ld;
const ld INF = 1e100; // for maximization set INF to 0 and negate costs

bool zero(ld x) {
    return fabs(x) < 1e-9; // change to x == 0 for integer types
}

struct hungarian {
    int n;
    vector<vector<ld> > cs;
    vector<int> matchl, matchr;
    hungarian(int _n, int _m) : n(max(_n, _m)), cs(n, vector<ld>(n)), matchl(n), matchr(n) {
        for(int x = 0; x < _n; x++)
            for(int y = 0; y < _m; y++)
                cs[x][y] = INF;
    }
    void set(int x, int y, ld c) {
        cs[x][y] = c;
    }
    ld assign() {
        int mat = 0;
        vector<ld> ds(n), y(n), z(n);
        vector<int> dad(n), vis(n);
        for(int i = 0; i < n; i++) {
            matchl[i] = matchr[i] = -1;
            y[i] = *min_element(cs[i].begin(), cs[i].end());
        }
        for(int j = 0; j < n; j++) {
            z[j] = cs[0][j] - y[0];
            for(int i = 1; i < n; i++)
                z[j] = min(z[j], cs[i][j] - y[i]);
        }
        for(int i = 0; i < n; i++)
            for(int j = 0; j < n; j++)
                if(matchr[j] == -1 && zero(cs[i][j] - y[i] - z[j])) {
                    matchl[i] = j;
                    matchr[j] = i;
                    mat++;
                    break;
                }
        for(;mat < n; mat++) {
            int s = 0, j, i;
            while(matchl[s] != -1) s++;
            for(int i = 0; i < n; i++) {
                dad[i] = -1;
                vis[i] = 0;
            }
            for(int k = 0; k < n; k++)
                ds[k] = cs[s][k] - y[s] - z[k];
            while(1) {
                j = -1;
                for(int k = 0; k < n; k++)
                    if(!vis[k] && (j == -1 || ds[k] < ds[j]))
                        j = k;
                vis[j] = 1;
                i = matchr[j];
                if(i == -1)
                    break;
                for(int k = 0; k < n; k++) if(!vis[k]) {
                    auto new_ds = ds[j] + cs[i][k] - y[i] - z[k];
                    if(ds[k] > new_ds) {
                        ds[k] = new_ds;
                        dad[k] = j;
                    }
                }
            }
            for(int k = 0; k < n; k++) if(k != j && vis[k]) {
                auto w = ds[k] - ds[j];
                z[k] += w;
                y[matchr[k]] -= w;
            }
            y[s] += ds[j];
            while(dad[j] != -1) {
                matchl[matchr[j] = matchr[dad[j]]] = j;
                j = dad[j];
            }
            matchr[j] = s;
            matchl[s] = j;
        }
        ld value = 0;
        for(int i = 0; i < n; i++)
            value += cs[i][matchl[i]];
        return value;
    }
};
```

stable_matching.cpp

```
/* Solves the rural hospital version of the stable matching problem.
```

```
 * This is, you have a set of doctors and hospitals and you want to
 * assign at most cap[i] doctors to the i-th hospital and each doctor
 * to at most one hospital. Also, each doctor has a list of candidate
 * hospitals, sorted in decreasing order of preference, and each hospital
 * has a list of candidate doctors, sorted in decreasing order of
 * preference too. Find an assigment of doctors to hospitals such that
 * there is no doctor D assigned to hospital H in which *both* D and H
 * would rather be matched with other candidates (consider the 'empty'
 * assignment as the least prefered one).
 * Remark: any stable matching has the same set of doctors assigned to
 * any hospital, and the same capactity used in every hospital.
 *
 * For the less constrained version of the problem, just set all the
 * capacities to 1.
 *
 * n - (input) number of doctors.
 * m - (input) number of hospitals.
 * cap - (input) array of hospitals' capacities. In the end of the algorithm,
       it will have the remaining capacity of each hospital.
 * doc - (input) list of hospital candidates of each doctor sorted by preference.
 * hos - (input) list of doctor candidates of each hospital sorted by preference.
 * assign - assign[i] is the hospital assigned to doctor i or -1 if it hasn't any
 *          hospital assigned.
 *
 * Complexity: O(L), where L is the total size of the lists of candidates.
 */

int n, m;
vector<int> doc[N], hos[N];
int pos[N], cap[N], last[N];
unordered_map<int, int> prior[N];

vector<int> galeShapley() {
    for(int i = 0; i < m; i++) {
        last[i] = sz(hos[i]) - 1;
        for(int j = 0; j < sz(hos[i]); j++)
            prior[i][hos[i][j]] = j;
    }
    vector<int> assign(n, -1);
    queue<int> Q;
    auto tryPush = [&] (int i) {
        if(pos[i] < sz(doc[i]))
            Q.push(i);
    };
    for(int i = 0; i < n; i++)
        tryPush(i);
    while(!Q.empty()) {
        int u = Q.front();
        Q.pop();
        int cur = doc[u][pos[u]++];
        if(cap[cur]) {
            cap[cur]--;
            assign[u] = cur;
        } else {
            if(prior[cur][u] <= last[cur]) {
                assign[u] = cur;
                while(assign[hos[cur][last[cur]]] != cur)
                    last[cur]--;
                assign[hos[cur][last[cur]]] = -1;
                tryPush(hos[cur][last[cur]]);
            } else tryPush(u);
        }
    }
    return assign;
}
```

flow_with_demands.cpp

```
/* Finds *any* feasible solution (if exists) to a flow network with demands (also known as
 * lower bounds). It's possible to find the minimum flow solution doing binary search with
 * a minor change.
 * It only does a graph transformation and applies a max flow algorithm on it, thus this
 * template depends of another flow algorithm template. In this example, that algorithm
 * is minimum-cost maximum-flow, but that's not always necessary. Adapting this template
 * to other flow algorithms should be easy.
 *
 * Constructor:
 * demands(n, s, t)
 * n - number of nodes in the original flow network.
 * s - source of the original flow network.
 * t - sink of the original flow network.
 *
 * Methods:
 * - addEdge(u, v, cap, dem, cost)
 *   adds an edge to the original flow network from u to v with capacity `cap`, flow
 *   demand `dem` and cost `cost`. For problems without cost, the last parameter should
 *   be erased.
 *
 * - getFlow()
 *   finishes building the auxiliary graph and returns the result of applying the max flow
 *   algorithm on it. If and only if the maximum flow is exactly equal to the sum of all
 *   demands, then exists a feasible solution to this problem.
 *
 * Complexity: depends on the flow algorithm used, but take into account that this method
 *             adds O(V) edges to the network flow.
 */

struct demands {
    vector<ll> din, dout;
    mcf g;
    int n, s, t;
    ll base = 0;
    demands(int n, int _s, int _t) : n(n), s(n), t(n + 1), din(n, 0), dout(n, 0) {
        g = mcf(n + 2, s, t);
        g.addEdge(_t, _s, INF, 0); // for minimum solution, change INF according to the value
                                   // of the binary search
    };
    void addEdge(int u, int v, ll cap, ll dem, ll cost) {
        din[v] += dem;
        dout[u] += dem;
        base += dem * cost;
        g.addEdge(u, v, cap - dem, cost);
    }
    pair<ll, ll> getFlow() {
        for(int i = 0; i < n; i++) {
            if(din[i] > 0) g.addEdge(s, i, din[i], 0);
            if(dout[i] > 0) g.addEdge(i, t, dout[i], 0);
        }
        auto ans = g.getFlow();
        return {ans.first, base + ans.second};
    }
};
```

min_cost_max_flow_dijkstra.cpp

```
/* Solves the minimum-cost maximum-flow problem using dijkstra for finding the incremental
 * shortest paths.
 *
 * Constructor:
 * mcf(n, s, t)
 * n - number of nodes in the flow graph.
 * s - source of the flow graph.
 * t - sink of the flow graph.
 *
 * Methods:
 * - addEdge(u, v, cap, cost)
 *   adds a directed edge from u to v with capacity `cap` and cost `cost`.
 * - getFlow()
 *   returns a pair of integers in which the first value is the maximum flow and the
 *   second is the minimum cost to achieve this flow.
 *
 * Complexity: There are two upper bounds to the time complexity of getFlow
 *              - O(max_flow * (E log V))
 *              - O(V * E * (E log V))
 */

const ll INF = 1LL << 60;

struct mcf {
    int n, s, t;
    ll cost, fl;
    vector<int> first, prev;
    vector<ll> dist;
    struct edge {
        int to, next;
        ll cap, cost;
        edge(int _to, ll _cap, ll _cost, int _next)
            : to(_to), cap(_cap), cost(_cost), next(_next) {};
    };
    vector<edge> g;
    mcf() {}
    mcf(int _n, int _s,int _t) : n(_n), s(_s), t(_t), fl(0), cost(0) {
        first.resize(n, -1);
        dist.resize(n);
        prev.resize(n);
        g.reserve(n*n);
    };
    void addEdge(int u, int v, ll cap, ll cost) {
        g.pb(edge(v, cap, cost, first[u]));
        first[u] = sz(g) - 1;
        g.pb(edge(u, 0, -cost, first[v]));
        first[v] = sz(g) - 1;
    }
    bool augment() {
        dist.assign(n, INF);
        dist[s] = 0;
        priority_queue<pair<ll, int> > q;
        q.push({0, s});
        while(!q.empty()) {
            if(dist[t] < INF) break;
            ll d, u;
            tie(d, u) = q.top();
            d *= -1;
            q.pop();
            if(dist[u] < d) continue;
            for(int e = first[u]; e != -1; e = g[e].next) {
                int v = g[e].to;
                ll ndist = d + g[e].cost;
                if(g[e].cap > 0 && ndist < dist[v]) {
                    dist[v] = ndist;
                    q.push({-ndist, v});
                    prev[v] = e;
                }
            }
        }
        return dist[t] < INF;
    }
    pair<ll, ll> getFlow() {
        while(augment()) {
            ll cur = t, curf = INF;
            while(cur != s) {
                int e = prev[cur];
                curf = min(curf, g[e].cap);
                cur = g[e^1].to;
            }
            fl += curf;
            cost += dist[t] * curf;
            cur = t;
            while(cur != s) {
                int e = prev[cur];
                g[e].cap -= curf;
                g[e^1].cap += curf;
                cur = g[e^1].to;
            }
        }
        return {fl, cost};
    }
};
```

min_cost_max_flow_spfa.cpp

```
/* Solves the minimum-cost maximum-flow problem using spfa for the finding the incremental
 * shortest paths. Useful when the edges costs are negative.
 *
 * Constructor:
 * mcf(n, s, t)
 * n - number of nodes in the flow graph.
 * s - source of the flow graph.
 * t - sink of the flow graph.
 *
 * Methods:
 * - addEdge(u, v, cap, cost)
 *   adds a directed edge from u to v with capacity `cap` and cost `cost`.
 * - getFlow()
 *   returns a pair of integers in which the first value is the maximum flow and the
 *   second is the minimum cost to achieve this flow.
 *
 * Complexity: There are two upper bounds to the time complexity of getFlow
 *             - O(max_flow * (E log V))
 *             - O(V * E * (E log V))
 */

const ll INF = 1LL << 60;

struct mcf {
    int n, s, t;
    ll cost, fl;
    vector<int> first, prev;
    vector<ll> dist;
    vector<bool> queued;
    struct edge {
        int to, next;
        ll cap, cost;
        edge(int _to, ll _cap, ll _cost, int _next)
            : to(_to), cap(_cap), cost(_cost), next(_next) {};
    };
    vector<edge> g;
    mcf() {}
    mcf(int _n, int _s,int _t) : n(_n), s(_s), t(_t), fl(0), cost(0) {
        queued.resize(n, 0);
        first.resize(n, -1);
        dist.resize(n);
        prev.resize(n);
        g.reserve(n*n);
    };
    void addEdge(int u, int v, ll cap, ll cost) {
        g.pb(edge(v, cap, cost, first[u]));
        first[u] = sz(g) - 1;
        g.pb(edge(u, 0, -cost, first[v]));
        first[v] = sz(g) - 1;
    }
    bool augment() {
        dist.assign(n, INF);
        dist[s] = 0;
        queued[s] = 1;
        queue<int> q;
        q.push(s);
        while(!q.empty()) {
            int u = q.front();
            q.pop();
            queued[u] = 0;
            for(int e = first[u]; e != -1; e = g[e].next) {
                int v = g[e].to;
                ll ndist = dist[u] + g[e].cost;
                if(g[e].cap > 0 && ndist < dist[v]) {
                    dist[v] = ndist;
                    prev[v] = e;
                    if(!queued[v]) {
                        q.push(v);
                        queued[v] = 1;
                    }
                }
            }
        }
        return dist[t] < INF;
    }
    pair<ll, ll> getFlow() {
        while(augment()) {
            ll cur = t, curf = INF;
            while(cur != s) {
                int e = prev[cur];
                curf = min(curf, g[e].cap);
                cur = g[e^1].to;
            }
            fl += curf;
            cost += dist[t] * curf;
            cur = t;
            while(cur != s) {
                int e = prev[cur];
                g[e].cap -= curf;
                g[e^1].cap += curf;
                cur = g[e^1].to;
            }
        }
        return {fl, cost};
    }
};
```

Geometry

halfplane_intersection.cpp, convex_hull.cpp, point_integer.cpp, delaunay.cpp, line_integer.cpp, angular_sweep.cpp, polygon_double.cpp, line_double.cpp, circle.cpp, polygon_integer.cpp, shamos_hoey.cpp, point_double.cpp

halfplane_intersection.cpp

```
/* Half-plane intersection algorithm. The result of intersecting half-planes is either
 * empty or a convex polygon (maybe degenerated). This template depends on point_double.cpp
 * and line_double.cpp.
 *
 * h - (input) set of half-planes to be intersected. Each half-plane is described as a pair
 * of points such that the half-plane is at the left of them.
 * pol - the intersection of the half-planes as a vector of points. If not empty, these
 * points describe the vertices of the resulting polygon in clock-wise order.
 * WARNING: Some points of the polygon might be repeated. This may be undesirable in some
 * cases but it's useful to distinguish between empty intersections and degenerated
 * polygons (such as a point, line, segment or half-line).
 *
 * Time complexity: O(n logn)
 */

struct halfplane: public line {
    ld ang;
    halfplane() {}
    halfplane(point _p, point _q) {
        p = _p; q = _q;
        point vec(q - p);
        ang = atan2(vec.y, vec.x);
    }
    bool operator <(const halfplane& other) const {
        if (fabsl(ang - other.ang) < EPS) return right(p, q, other.p);
            return ang < other.ang;
    }
    bool operator ==(const halfplane& other) const {
        return fabsl(ang - other.ang) < EPS;
    }
    bool out(point r) {
        return right(p, q, r);
    }
};

vector<point> hp_intersect(vector<halfplane> h) {
    point box[4] = {{-INF, -INF}, {INF, -INF}, {INF, INF}, {-INF, INF}};
    for(int i = 0; i < 4; i++)
        h.pb(halfplane(box[i], box[(i+1) % 4]));
    sort(h.begin(), h.end());
    h.resize(unique(h.begin(), h.end()) - h.begin());
    deque<halfplane> dq;
    for(auto hp: h) {
        while(sz(dq) > 1 && hp.out(intersect(dq.back(), dq[sz(dq) - 2])))
            dq.pop_back();
        while(sz(dq) > 1 && hp.out(intersect(dq[0], dq[1])))
            dq.pop_front();
        dq.pb(hp);
    }
    while(sz(dq) > 2 && dq[0].out(intersect(dq.back(), dq[sz(dq) - 2])))
        dq.pop_back();
    while(sz(dq) > 2 && dq.back().out(intersect(dq[0], dq[1])))
        dq.pop_front();
    if(sz(dq) < 3) return {};
    vector<point> pol(sz(dq));
    for(int i = 0; i < sz(dq); i++) {
        pol[i] = intersect(dq[i], dq[(i+1) % sz(dq)]);
    }
    return pol;
}
```

convex_hull.cpp

```
/* Finds the convex hull of a given set of points. This templates requires
 * the struct point defined in point_integer.cpp or in point_double.cpp
 *
 * p - (input) vector of points for which the convex hull will be found.
 * ch - convex hull of `p` in counter-clockwise order.
 */

vector<point> convexHull(vector<point> p) {
    int n = sz(p);
    sort(p.begin(), p.end());
    vector<point> low, up;
    for(int i = 0; i < n; i++) {
        if(i && p[i] == p[i - 1]) continue;
        while(sz(up) >= 2 && !right(up[sz(up)-2], up.back(), p[i]))
            up.pop_back();
        up.pb(p[i]);
        while(sz(low) >= 2 && !left(low[sz(low)-2], low.back(), p[i]))
            low.pop_back();
        low.pb(p[i]);
    }
    vector<point> ch;
    if(sz(low) == 1) return low;
    for(int i = 0; i < sz(low) - 1; i++)
        ch.pb(low[i]);
    for(int i = sz(up) - 1; i >= 1; i--)
        ch.pb(up[i]);
    return ch;
}
```

point_integer.cpp

```
/* Basic structure of point and operations related with it. This template assumes
 * integer coordinates.
 *
 * All operations' time complexity are O(1)
 */

struct point {
    ll x, y;
    point(ll x, ll y) : x(x), y(y) {}
    point() {}
    ll norm2() {
        return *this * *this;
    }
    bool operator==(const point& other) const {
        return x == other.x && y == other.y;
    }
    point operator+(const point& other) const {
        return point(x + other.x, y + other.y);
    }
    point operator-(const point& other) const {
        return point(x - other.x, y - other.y);
    }
    point operator*(ll t) const {
        return point(x * t, y * t);
    }
    point operator/(ll t) const {
        return point(x / t, y / t);
    }
    ll operator*(const point& other) const {
        return x*other.x + y*other.y;
    }
    ll operator^(const point& other) const { // cross product
        return x*other.y - y*other.x;
    }
    bool operator<(const point& other) const { // for sweep line
        return x < other.x || (x == other.x && y < other.y);
    }
    point rotate(point r) {
        return point(*this ^ r, *this * r);
    }
};
point ccw90(1, 0);
point cw90(-1, 0);

ll dist2(point p, point q) { // squared distance
    return (p - q).norm2();
}

ll area2(point a, point b, point c) { // two times signed area of triangle abc
    return (b - a) ^ (c - a);
}

bool left(point a, point b, point c) {
    return area2(a, b, c) > 0;
}

bool right(point a, point b, point c) {
    return area2(a, b, c) < 0;
}

bool collinear(point a, point b, point c) {
    return abs(area2(a, b, c)) == 0;
}

// Returns 0 if vectors a and b are not parallel.
// If they are parallel, returns 1 if they have the same direction
// and returns -1 otherwise
int paral(point a, point b) {
    if((a ^ b) != 0) return 0;
    if((a.x > 0) == (b.x > 0) && (a.y > 0) == (b.y > 0))
        return 1;
    return -1;
}
```

delaunay.cpp

```
/* Builds a delaunay triangulation of a given set of points. This templates requires
 * the struct point defined in point_integer.cpp
 *
 * p - (input) vector of points for which the convex hull will be found.
 * adj - the delaunay triangulation of `p` as an adjacency list in which every vertex
 *       has its original index in `p`
 */

struct quadEdge {
    point o;
    quadEdge *rot, *nxt;
    bool used;
    quadEdge(point o = point(INF, INF))
        : o(o), rot(nullptr), nxt(nullptr), used(false) {}
    quadEdge* rev() const {
        return rot->rot;
    }
    quadEdge* lnext() const {
        return rot->rev()->nxt->rot;
    }
    quadEdge* prev() const {
        return rot->nxt->rot;
    }
    point dest() const {
        return rev()->o;
    }
};

quadEdge* makeEdge(point from, point to) {
    vector<quadEdge*> e(4);
    e[0] = new quadEdge(from);
    e[1] = new quadEdge(to);
    e[2] = new quadEdge; e[3] = new quadEdge;
    tie(e[0]->rot, e[1]->rot, e[2]->rot, e[3]->rot) = {e[2], e[3], e[1], e[0]};
    tie(e[0]->nxt, e[1]->nxt, e[2]->nxt, e[3]->nxt) = {e[0], e[1], e[3], e[2]};
    return e[0];
}

void splice(quadEdge* a, quadEdge* b) {
    swap(a->nxt->rot->nxt, b->nxt->rot->nxt);
    swap(a->nxt, b->nxt);
}

void deleteEdge(quadEdge* &e, quadEdge* ne) {
    splice(e, e->prev());
    splice(e->rev(), e->rev()->prev());
    delete e->rev()->rot;
    delete e->rev();
    delete e->rot;
    delete e;
    e = ne;
}

quadEdge* connect(quadEdge* a, quadEdge* b) {
    quadEdge* e = makeEdge(a->dest(), b->o);
    splice(e, a->lnext());
    splice(e->rev(), b);
    return e;
}
```

```
__int128 det3(point a, point b, point c) {
    vector<__int128> len = {a.norm2(), b.norm2(), c.norm2()};
    return a.x * (b.y * len[2] - c.y * len[1])
            - a.y * (b.x * len[2] - c.x * len[1])
            + len[0] * (b ^ c);
}

bool inCircle(point a, point b, point c, point d) {
    __int128 det = -det3(b, c, d);
    det += det3(a, c, d);
    det -= det3(a, b, d);
    det += det3(a, b, c);
    return det > 0;
}

pair<quadEdge*, quadEdge*> buildTr(int l, int r, vector<point>& p) {
    if(r - l <= 3) {
        quadEdge* a = makeEdge(p[l], p[l + 1]), *b = makeEdge(p[l + 1], p[r - 1]);
        if(r - l == 2) return mp(a, a->rev());
        splice(a->rev(), b);
        ll sg = area2(p[l], p[l + 1], p[l + 2]);
        quadEdge* c = sg ? connect(b, a) : 0;
        if(sg >= 0) return mp(a, b->rev());
        else return mp(c->rev(), c);
    }
    int m = (l + r) >> 1;
    quadEdge *ldo, *ldi, *rdo, *rdi;
    tie(ldo, ldi) = buildTr(l, m, p);
    tie(rdi, rdo) = buildTr(m, r, p);
    while(1) {
        if(left(rdi->o, ldi->o, ldi->dest()))
            ldi = ldi->lnext();
        else if(right(ldi->o, rdi->o, rdi->dest()))
            rdi = rdi->rev()->nxt;
        else break;
    }
    quadEdge* basel = connect(rdi->rev(), ldi);
    auto valid = [&](quadEdge* e) {
        return right(e->dest(), basel->o, basel->dest());
    };
    if(ldi->o == ldo->o) ldo = basel->rev();
    if(rdi->o == rdo->o) rdo = basel;
    while(1) {
        quadEdge *lcand = basel->rev()->nxt;
        if(valid(lcand)) {
            while(inCircle(basel->dest(), basel->o, lcand->dest(), lcand->nxt->dest()))
                deleteEdge(lcand, lcand->nxt);
        }
        quadEdge *rcand = basel->prev();
        if(valid(rcand)) {
            while(inCircle(basel->dest(), basel->o, rcand->dest(), rcand->prev()->dest()))
                deleteEdge(rcand, rcand->prev());
        }
        if(!valid(lcand) && !valid(rcand))
            break;
        if(!valid(lcand)
            || (valid(rcand) && inCircle(lcand->dest(), lcand->o, rcand->o, rcand->dest())))
            basel = connect(rcand, basel->rev());
        else basel = connect(basel->rev(), lcand->rev());
    }
    return mp(ldo, rdo);
}

void delaunay(vector<point> p, vector<vector<int>>& adj) {
    vector<point> temp = p;
    map<point, int> m;
    for(int i = 0; i < sz(p); i++) m[p[i]] = i;
    sort(p.begin(), p.end());
    adj.resize(sz(p));
    auto add_edge = [&](point a, point b) {
            adj[m[a]].pb(m[b]);
    };
    bool col = 1;
    for(int i = 2; i < sz(p); i++) col &= collinear(p[0], p[1], p[i]);
    if(col) {
        for(int i = 0; i + 1 < sz(p); i++) {
            add_edge(p[i], p[i + 1]);
            add_edge(p[i + 1], p[i]);
        }
    } else {
        quadEdge* e = buildTr(0, sz(p), p).first;
        vector<quadEdge*> edges = {e};
        for(int i = 0; i < sz(edges); e = edges[i++]) {
            for(quadEdge* at = e; !at->used; at = at->nxt) {
                at->used = 1;
                add_edge(at->o, at->rev()->o);
                edges.pb(at->rev());
            }
        }
    }
}
```

line_integer.cpp

```
/* Basic structure of line defined by two 2D points which the line goes through.
 * Some of the functions assume that the line is in fact a segment with the two
 * points as its endpoints. Those operations are preceded by a comment stating
 * that this is the case.
 *
 * This template depends on point_integer.cpp and works only with integers.
 *
 * All operations' time complexity are O(1)
 */

struct line {
    point p, q;
    line(point p, point q) : p(p), q(q) {}
    line() {}
    bool has(const point& r) const {
        return paral((r - p), (q - p));
    }
    bool operator==(const line& other) const { // assumes that direction does not matter
        return has(other.p) && has(other.q);
    }
    bool isVert() {
        return p.x == q.x;
    }
    // the following operations are for segments only
    bool segHas(point r) {
        return collinear(p, q, r)
            && (min(p.x, q.x) <= r.x && r.x <= max(p.x, q.x))
            && (min(p.y, q.y) <= r.y && r.y <= max(p.y, q.y));
    }
    line rotate(point r) { // rotates segment pivoted in p
        return line(p, p + (q - p).rotate(r));
    }
    bool operator<(const line& other) const { // for Shamos-Hoey
        // the case when q == other.p is such that we cosider this == other.
        // that might give wrong answer, so be careful
        if(p == other.p) return left(p, q, other.q);
        if(!isVert() && (other.isVert() || p.x < other.p.x))
            return left(p, q, other.p);
        return left(p, other.q, other.p);
    }
};

int paraline(line a, line b) {
    return paral(a.q - a.p, b.q - b.p);
}

// the following functions are for segments only
bool checkInter(line a, line b) {
    if(a.segHas(b.p) || a.segHas(b.q) || b.segHas(a.p) || b.segHas(a.q))
        return 1;
    return left(a.p, a.q, b.p) != left(a.p, a.q, b.q)
            && left(b.p, b.q, a.p) != left(b.p, b.q, a.q);
}
```

angular_sweep.cpp

```
/* Version of point_integer with < operator for angular sweep.
 * The angle range is ]-\pi, \pi] and the template considers that the angle of the point
 * (0, 0) is 0. This template only contains a the operators needed for the for applying
 * the angular sort.
 */

struct point {
    ll x, y;
    point(ll x, ll y) : x(x), y(y) {}
    point() {}
    ll operator*(const point& other) const {
        return x*other.x + y*other.y;
    }
    ll operator^(const point& other) const { // cross product
```

```
        return x*other.y - y*other.x;
    }
    int side() const {
        return y > 0 || (y == 0 && x < 0);
    }
    bool operator==(const point& other) const {
        return x == other.x && y == other.y;
    }
    bool operator<(const point& other) const { // for angular sweep
        int this_side = side(), other_side = other.side();
        if(this_side != other_side) return this_side < other_side;
        if(*this == point(0, 0)) return 0;
        if(other == point(0, 0)) return 1;
        return (*this ^ other) > 0;
    }
};
```

polygon_double.cpp

```
/* Basic structure of polygon.
 *
 * This template depends on point_double.cpp since it can work with double coordinates.
 *
 * All operations' time complexity are O(1) unless stated otherwise.
 */

struct polygon {
    vector<point> p;
    int n;
    polygon() : n(0) {}
    polygon(vector<point> _p) {
        p = _p;
        n = sz(p);
    }
    void add(point q) {
        p.pb(q);
        n++;
    }
    // If positive, the polygon is in ccw order. It is in cw order otherwise.
    ld orientation() { // O(n)
        ld acum = 0;
        for(int i = 0; i < n; i++)
            acum += p[i] ^ p[(i + 1) % n];
        return acum;
    }
    ld area() { // O(n)
        return abs(orientation()) / 2.0;
    }
    void turnCcw() { // O(n)
        if(orientation() < -EPS)
            reverse(p.begin(), p.end());
    }
    bool has(point q) { // O(log n). The polygon must be convex and in ccw order.
        if(right(p[0], p[1], q) || left(p[0], p[n-1], q)) return 0;
        int lo = 1, hi = n;
        while(lo + 1 < hi) {
            int mid = (lo + hi) >> 1;
            if(!right(p[0], p[mid], q)) lo = mid;
            else hi = mid;
        }
        return hi != n ? !right(p[lo], p[hi], q) : dist(p[0], q) < dist2(p[0], p[n-1]) + EPS;
    }
    ld calipers() { // O(n). The polygon must be convex and in ccw order.
        ld ans = 0;
        for(int i = 0, j = 1; i < n; i++) {
            point vec_i = p[(i+1)%n] - p[i];
            while((vec_i ^ (p[(j+1)%n] - p[j])) > EPS)
                j = (j + 1) % n;
            ans = max(ans, dist(p[i], p[j])); // Example with polygon diameter
        }
        return ans;
    }
    int extreme(const function<bool(point, point)> &cmp) {
        auto isExtreme = [&](int i, bool& curDir) -> bool {
            curDir = cmp(p[(i + 1) % n], p[i]);
            return !cmp(p[(i + n - 1) % n], p[i]) && !curDir;
        };
        bool lastDir, curDir;
        if(isExtreme(0, lastDir)) return 0;
        int lo = 0, hi = n;
        while(lo + 1 < hi) {
            int m = (lo + hi) >> 1;
            if(isExtreme(m, curDir)) return m;
            bool relDir = cmp(p[m], p[lo]);
            if((!lastDir && curDir) || (lastDir == curDir && relDir == curDir)) {
                lo = m;
                lastDir = curDir;
            } else hi = m;
        }
        return lo;
    }
    pair<int, int> tangent(point q) { // O(log n) for convex polygon in ccw orientation
        // Finds the indices of the two tangents to an external point q
        auto leftTangent = [&](point r, point s) -> bool {
            return right(q, r, s);
        };
        auto rightTangent = [&](point r, point s) -> bool {
            return left(q, r, s);
        };
        return {extreme(leftTangent), extreme(rightTangent)};
    }
    int maximize(point v) { // O(log n) for convex polygon in ccw orientation
        // Finds the extreme point in the direction of the vector
        return extreme([&](point p, point q) {return p * v > q * v + EPS;});
    }
    void normalize() { // p[0] becomes the lowest leftmost point
        rotate(p.begin(), min_element(p.begin(), p.end()), p.end());
    }
    polygon operator+(polygon& other) { // Minkowsky sum
        vector<point> sum;
        normalize();
        other.normalize();
        ld dir;
        for(int i = 0, j = 0; i < n || j < other.n; i += dir > -EPS, j = dir < EPS) {
            sum.pb(p[i % n] + other.p[j % other.n]);
            dir = (p[(i + 1) % n] - p[i % n])
                    ^ (other.p[(j + 1) % other.n] - other.p[j % other.n]);
        }
        return polygon(sum);
    }
};
```

line_double.cpp

```
/* Basic structure of line defined by two 2D points which the line goes through.
 * Some of the functions assume that the line is in fact a segment with the two
 * points as its endpoints. Those operations are preceded by a comment stating
 * that this is the case.
 *
 * This template depends on point_double.cpp and hence the coordinates don't need
 * to be integers.
 *
 * All operations' time complexity are O(1)
 */

struct line {
    point p, q;
    line(point p, point q) : p(p), q(q) {}
    line() {}
    bool has(point r) const {
        return paral((r - p), (q - p));
    }
    bool operator==(const line& other) const { // assumes that direction does not matter
        return has(other.p) && has(other.q);
    }
    bool isVert() {
        return abs(p.x - q.x) <= EPS;
    }
    point proj(point r) {
        point q_vec = q - p, r_vec = r - p;
        return p + q_vec * (q_vec * r_vec / q_vec.norm2());
    }
    ld dist(point r) {
        return (r - proj(r)).norm();
    }
    // the following operations are for segments only
    bool segHas(point r) {
        return collinear(p, q, r)
            && (min(p.x, q.x) < r.x + EPS && r.x < max(p.x, q.x) + EPS)
            && (min(p.y, q.y) < r.y + EPS && r.y < max(p.y, q.y) + EPS);
    }
    line rotate(point r) { // rotates segment pivoted in p
```

```
        return line(p, p + (q - p).rotate(r));
    }
    line rotate(ld ang) { // rotates segment pivoted in p by angle
        return line(p, p + (q - p).rotate(ang));
    }
};

int paraline(line a, line b) {
    return paral(a.q - a.p, b.q - b.p);
}

point intersect(line a, line b) {
    if(paraline(a, b)) return point(INF, INF);
    point v_a = (a.q - a.p), v_b = (b.q - b.p);
    ld c_a = v_a ^ a.p, c_b = v_b ^ b.p;
    return (v_b*c_a - v_a*c_b) / (v_a ^ v_b);
}

// the following functions are for segments only
bool checkInter(line a, line b) {
    if(a.segHas(b.p) || a.segHas(b.q) || b.segHas(a.p) || b.segHas(a.q))
        return 1;
    return left(a.p, a.q, b.p) != left(a.p, a.q, b.q)
            && left(b.p, b.q, a.p) != left(b.p, b.q, a.q);
}
```

circle.cpp

```
/* Basic structure of circle and operations related with it. This template works
* only with double numbers since most of the operations of a circle can't be
* done with only integers. Therefore, this template depends on point_double.cpp.
*
* All operations' time complexity are O(1)
*/

const ld PI = acos(-1);

struct circle {
    point o; ld r;
    circle() {}
    circle(point o, ld r) : o(o), r(r) {}
    bool has(point p) {
        return (o - p).norm2() < r*r + EPS;
    }
    vector<point> operator/(circle c) { // Intersection of circles.
        vector<point> inter;                 // The points in the output are in ccw order.
        ld d = (o - c.o).norm();
        if(r + c.r < d - EPS || d + min(r, c.r) < max(r, c.r) - EPS)
            return {};
        ld x = (r*r - c.r*c.r + d*d) / (2*d);
        ld y = sqrt(r*r - x*x);
        point v = (c.o - o) / d;
        inter.pb(o + v*x + v.rotate(cw90)*y);
        if(y > EPS) inter.pb(o + v*x + v.rotate(ccw90)*y);
        return inter;
    }
    vector<point> tang(point p){
        ld d = sqrt((p - o).norm2() - r*r);
        return *this / circle(p, d);
    }
    bool in(circle c){ // non strictly inside
        ld d = (o - c.o).norm();
        return d + r < c.r + EPS;
    }
};
```

polygon_integer.cpp

```
/* Basic structure of polygon.
 *
 * This template depends on point_integer.cpp since it only works for integer coordinates.
 *
 * All operations' time complexity are O(1) unless stated otherwise.
 */

struct polygon {
    vector<point> p;
    int n;
    polygon() : n(0) {}
    polygon(vector<point> _p) {
        p = _p;
        n = sz(p);
    }
    void add(point q) {
        p.pb(q);
        n++;
    }
    // If positive, the polygon is in ccw order. It is in cw order otherwise.
    ll orientation() { // O(n)
        ll acum = 0;
        for(int i = 0; i < n; i++)
            acum += p[i] ^ p[(i + 1) % n];
        return acum;
    }
    ll area2() { // O(n)
        return abs(orientation());
    }
    void turnCcw() { // O(n)
        if(orientation() < 0)
            reverse(p.begin(), p.end());
    }
    bool has(point q) { // O(log n). The polygon must be convex and in ccw order
        if(right(p[0], p[1], q) || left(p[0], p[n-1], q)) return 0;
        int lo = 1, hi = n;
        while(lo + 1 < hi) {
            int mid = (lo + hi) >> 1;
            if(!right(p[0], p[mid], q)) lo = mid;
            else hi = mid;
        }
        return hi != n ? !right(p[lo], p[hi], q) : dist2(p[0], q) <= dist2(p[0], p[n-1]);
    }
    ll calipers() { // O(n). The polygon must be convex and in ccw order.
        ll ans = 0;
        for(int i = 0, j = 1; i < n; i++) {
            point vec_i = p[(i+1)%n] - p[i];
            while((vec_i ^ (p[(j+1)%n] - p[j])) > 0)
                j = (j + 1) % n;
            ans = max(ans, dist2(p[i], p[j])); // Example with polygon diameter squared
        }
        return ans;
    }
    int extreme(const function<bool(point, point)> &cmp) {
        auto isExtreme = [&](int i, bool& curDir) -> bool {
            curDir = cmp(p[(i + 1) % n], p[i]);
            return !cmp(p[(i + n - 1) % n], p[i]) && !curDir;
        };
        bool lastDir, curDir;
        if(isExtreme(0, lastDir)) return 0;
        int lo = 0, hi = n;
        while(lo + 1 < hi) {
            int m = (lo + hi) >> 1;
            if(isExtreme(m, curDir)) return m;
            bool relDir = cmp(p[m], p[lo]);
            if((!lastDir && curDir) || (lastDir == curDir && relDir == curDir)) {
                lo = m;
                lastDir = curDir;
            } else hi = m;
        }
        return lo;
    }
    pair<int, int> tangent(point q) { // O(log n) for convex polygon in ccw orientation
        // Finds the indices of the two tangents to an external point q
        auto leftTangent = [&](point r, point s) -> bool {
            return right(q, r, s);
        };
        auto rightTangent = [&](point r, point s) -> bool {
            return left(q, r, s);
        };
        return {extreme(leftTangent), extreme(rightTangent)};
    }
    int maximize(point v) { // O(log n) for convex polygon in ccw orientation
        // Finds the extreme point in the direction of the vector
        return extreme([&](point p, point q) {return p * v > q * v;});
    }
    void normalize() { // p[0] becomes the lowest leftmost point
        rotate(p.begin(), min_element(p.begin(), p.end()), p.end());
    }
    polygon operator+(polygon& other) { // Minkowsky sum
        vector<point> sum;
        normalize();
        other.normalize();
        for(ll i = 0, j = 0, dir; i < n || j < other.n; i += dir >= 0, j += dir <= 0) {
            sum.pb(p[i % n] + other.p[j % other.n]);
            dir = (p[(i + 1) % n] - p[i % n])
```

```
                    ^ (other.p[(j + 1) % other.n] - other.p[j % other.n]);
        }
        return polygon(sum);
    }
};
```

shamos_hoey.cpp

```
/* Shamos-Hoey algorithm for checking wether a collection of segments have an intersection.
 * This template depends on point_integer.cpp  and line_integer.cpp.
 *
 * seg - (input) collection of segments.
 *
 * Time complexity: O(n logn)
 */

bool shamos_hoey(vector<line> seg) {
    // create sweep line events {x, type, seg_id}
    vector<array<ll, 3> > ev;
    for(int i = 0; i < sz(seg); i++) {
    if(seg[i].q < seg[i].p) swap(seg[i].p, seg[i].q);
        ev.pb({seg[i].p.x, 0, i});
        ev.pb({seg[i].q.x, 1, i});
    }
    sort(ev.begin(), ev.end());
    set<line> s;
    for(auto e: ev) {
        line at = seg[e[2]];
        if(!e[1]) {
            auto nxt = s.lower_bound(at);
            if((nxt != s.end() && checkInter(*nxt, at))
                || (nxt != s.begin() && checkInter(*(--nxt), at)))
                    return 1;
            s.insert(at);
        } else {
            auto nxt = s.upper_bound(at), cur = nxt, prev = --cur;
            if(nxt != s.end() && prev != s.begin()
                && checkInter(*nxt, *(--prev))) return 1;
            s.erase(cur);
        }
    }
    return 0;
}
```

point_double.cpp

```
/* Basic structure of point and operations related with it. This template works
 * with double coordinates.
 *
 * All operations' time complexity are O(1)
 */

typedef long double ld;
const ld EPS = 1e-9;

struct point {
    ld x, y;
    point(ld x, ld y) : x(x), y(y) {}
    point() {}
    ld norm2() {
        return *this * *this;
    }
    ld norm() {
        return sqrt(norm2());
    }
    bool operator==(const point& other) const {
        return abs(x - other.x) < EPS && abs(y - other.y) < EPS;
    }
    point operator+(const point& other) const {
        return point(x + other.x, y + other.y);
    }
    point operator-(const point& other) const {
        return point(x - other.x, y - other.y);
    }
    point operator*(ld t) const {
        return point(x * t, y * t);
    }
    point operator/(ld t) const {
        return point(x / t, y / t);
    }
    ld operator*(const point& other) const {
        return x*other.x + y*other.y;
    }
    ld operator^(const point& other) const { // cross product
        return x*other.y - y*other.x;
    }
    bool operator<(const point& other) const { // for sweep line
        return x < other.x - EPS || (abs(x - other.x) < EPS && y < other.y - EPS);
    }
    point rotate(point r) {
        return point(*this ^ r, *this * r);
    }
    point rotate(ld ang) {
        return rotate(point(sin(ang), cos(ang)));
    }
    ld angle(point& other) { // only works for angles in the range [0, PI]
        ld cos_val = min(1.0L, max(-1.0L, *this * other / (norm() * other.norm())));
        return acos(cos_val);
    }
};
point ccw90(1, 0);
point cw90(-1, 0);

ld dist2(point p, point q) { // squared distance
    return (p - q).norm2();
}

ld dist(point p, point q) {
    return sqrt(dist2(p, q));
}

ld area2(point a, point b, point c) { // two times signed area of triangle abc
    return (b - a) ^ (c - a);
}

bool left(point a, point b, point c) {
    return area2(a, b, c) > EPS;
}

bool right(point a, point b, point c) {
    return area2(a, b, c) < -EPS;
}

bool collinear(point a, point b, point c) {
    return abs(area2(a, b, c)) < EPS;
}

// Returns 0 if vectors a and b are not parallel.
// If they are parallel, returns 1 if they have the same direction
// and returns -1 otherwise
int paral(point a, point b) {
    if((a ^ b) != 0) return 0;
    if((a.x > EPS) == (b.x > EPS) && (a.y > EPS) == (b.y > EPS))
        return 1;
    return -1;
}
```

Graphs

manhattan_mst.cpp, lca.cpp, block_cut_tree.cpp, articulation_points.cpp, bridges.cpp, two_sat.cpp, tarjan_scc.cpp

manhattan_mst.cpp

```
/* Computes O(n) edges that contains all the edges of the  Manhattan MST
 * of a set of points int the 2D plane. Must apply Kruskal to these edges
 * to obtain the actual MST.
 *
 * xs - (input) coordinates in the x-axis
 * ys - (input) coordinates in the y-axis
 * E - superset of the manhattan MST edges
 *
 * Complexity: O(n logn)
 */

vector<pair<ll, pair<int, int>> > E;

void manhattan_mst(vector<ll> xs, vector<ll> ys){
    int n = sz(xs);
    vector<int> ord(n);
    for(int i = 0; i < n; i++)
        ord[i]=i;
    for(int s = 0; s < 2; s++) {
        for(int t = 0; t < 2; t++) {
            auto cmp = [&](int i, int j) -> bool {
                if(xs[i] + ys[i] == xs[j] + ys[j]) return ys[i] < ys[j];
                return xs[i] + ys[i] < xs[j] + ys[j];
            };
            sort(ord.begin(), ord.end(), cmp);
            map<int, int> id;
            for(int i: ord) {
                for(auto it = id.lower_bound(-ys[i]); it != id.end(); it = id.erase(it)) {
                    int j = it->ss;
                    if(xs[j] - ys[j] > xs[i] - ys[i]) break;
                    E.pb(mp(abs(xs[i] - xs[j]) + abs(ys[i] - ys[j]), mp(i, j)));
                }
                id[-ys[i]] = i;
            }
            swap(xs,ys);
        }
        for(int i = 0; i < n; i++)
            xs[i] *= -1;
    }
}
```

lca.cpp

```
/* Binary lifting algorithm to compute the lowest common ancestor of two nodes in a tree
 *
 * To use this template, first you need to add the (undirected) edges using `addEdge` and,
 * after all the edges has been added, call `eulerTour(root, root)` where `root` is the
 * root of the tree. Then you can use `lca(u, v)` to fin the lowest common ancestor of two
 * nodes `u` and `v`.
 *
 * LOGN - logarithm in base 2 of N
 * anc[u][j] - (output) 2^j-th ancestor o u
 *
 * Time complexity: O(n log n) for precomputing and O(log n) per query.
 * Space complexity: O(n log n)
 */

const int LOGN = 20;

int anc[N][LOGN], tam[N], tin[N];
vector<int> adj[N];
int node_id;

void addEdge(int u, int v) {
    adj[u].pb(v);
    adj[v].pb(u);
}
```

```
// Call this with eulerToor(root, root)
int eulerTour(int u, int p) {
    anc[u][0] = p;
    tam[u] = 1;
    tin[u] = node_id++;
    for(int i = 1; i < LOGN; i++)
        anc[u][i] = anc[anc[u][i-1]][i-1];
    for(int v: adj[u]) if(v != p)
        tam[u] += eulerTour(v, u);
    return tam[u];
}

// Check if u is ancestor of v in O(1)
bool isAnc(int u, int v) {
    return tin[u] <= tin[v] && tin[v] < tin[u] + tam[u];
}

int lca(int u, int v) {
    if(isAnc(u, v)) return u;
    if(isAnc(v, u)) return v;
    for(int i = LOGN - 1; i >= 0; i--)
        if(!isAnc(anc[u][i], v)) u = anc[u][i];
    return anc[u][0];
}
```

block_cut_tree.cpp

```
/* Builds the block-cut tree of a given graph. The block-cut tree is a graph
 * decomposition in biconnected components (blocks) and articulation points (cuts).
 * Every block is uniquely defined by a set of edges. This template considers
 * the the articulation point only belongs to the cut.
 * Usage: add undirected edges with addEdge(u, v) and then call build(n), when n is
 * the total number of vertices in the graph. WARNING: make sure to do not add
 * parallel edges. Also, notice that you have to clear adj, adj_bct and edge_cont
 * by yourself between different test cases.
 *
 * n - (input) number of vertices.
 * belong - belong[i] is the component in which the vertex i is in, either a block
          or a cut.
 * edge_belong - edge_belong[i] is the block in which the edge i is in.
 * ap - ap[i] is true if the vertex i is an articulation point. WARNING: this template
      considers isolated vertices as articulation points.
 * adj_bct - the list of adjacency of the block-cut tree
 *
 * Complexity: O(n) to find all biconnected components and articulation points.
               O(n log n) to build the block-cut tree.
 */

int edge_cont, comp, temp, stp;
pair<int, int> edge[N];
int belong[N], low[N], disc[N], st[N], edge_belong[N];
bool ap[N];
vector<int> adj[N], adj_bct[N];

void bcc(int v, int p=-1) {
    low[v] = disc[v] = ++temp;
    int child = 0;
    if(sz(adj[v]) == 0) ap[v] = 1;
    for(int ed: adj[v]) {
        int u = edge[ed].first ^ edge[ed].second ^ v;
        if(!disc[u]) {
            child++;
            st[stp++] = ed;
            bcc(u, v);
            low[v] = min(low[v], low[u]);
            if(low[u] >= disc[v]) {
                if(p != -1 || child > 1) ap[v] = 1;
                while(1) {
                    int e = st[--stp];
                    edge_belong[e] = comp;
                    belong[edge[e].first] = comp;
                    belong[edge[e].second] = comp;
                    if(e == ed) break;
                }
                comp++;
            }
        } else if(u != p) {
            low[v] = min(low[v], disc[u]);
            if(disc[u] < disc[v])
                st[stp++] = ed;
        }
    }
}

void addEdge(int u, int v) {
    adj[u].pb(edge_cont);
    adj[v].pb(edge_cont);
    edge[edge_cont++] = {u, v};
}

void build(int n) {
    for(int i = 0; i < n; i++) {
        low[i] = disc[i] = ap[i] = 0;
    }
    temp = comp = stp = 0;
    for(int i = 0; i < n; i++)
        if(!disc[i])
            bcc(i);
    for(int u = 0; u < n; i++) if(ap[u]) {
        for(int e: adj[u]) {
            adj_bct[comp].pb(edge_belong[e]);
            adj_bct[edge_belong[e]].pb(comp);
        }
        belong[u] = comp;
        comp++;
    }
    for(int i = 0; i < comp; i++) {
        sort(all(adj_bct[i]));
        adj_bct[i].resize(unique(all(adj_bct[i])) - adj_bct[i].begin());
    }
}
```

articulation_points.cpp

```
/* Finds the articulation points of an undirected graph. The edges might be parallel or
 * self-loops without problems.
 *
 * Usage: first add the edges with `addEdge(u, v)`, and then call `findAps(n)`.
 *
 * n - (input) number of vertices.
 * ap - ap[i] is true if the vertex i is an articulation point.
 *
 * Complexity: O(V + E) time and space
 */

int low[N], disc[N], t_in;
bool ap[N];
vector<int> adj[N];

void compute(int v, int root, int p=-1) {
    low[v] = disc[v] = ++t_in;
    int childs = 0;
    for(int u : adj[v]) {
        if(!disc[u]) {
            childs++;
            compute(u, root, v);
            low[v] = min(low[v], low[u]);
            if(low[u] >= disc[v] && (v != root || childs > 1))
                ap[v] = 1;
        } else if(u != p) {
            low[v] = min(low[v], disc[u]);
        }
    }
}

void addEdge(int u, int v) {
    adj[u].pb(v);
    adj[v].pb(u);
}

void findAps(int n) {
    for(int i = 0; i < n; i++) {
        low[i] = disc[i] = ap[i] = 0;
    }
    t_in = 0;
    for(int i = 0; i < n; i++) {
        if(!disc[i]) compute(i, i);
    }
}

// Must be call between testcases
void clearGraph(int n) {
    for(int i = 0; i < n; i++)
```

```
        adj[i].clear();
}
```

bridges.cpp

```
/* Finds the bridges of an undirected graph. The edges might be parallel or self-loops
 * without problems.
 *
 * Usage: first add the edges with `addEdge(u, v)`, and then call `findBridges(n)`
 *
 * n - (input) number of vertices.
 * bridge - bridge[i] is true if the i-th edge is a bridge.
 *
 * Complexity: O(V + E) time and space
 */

vector<pair<int, int>> edge;
int low[N], disc[N], t_in;
bool bridge[N];
vector<int> adj[N];

void compute(int v, int p=-1) {
    low[v] = disc[v] = ++t_in;
    for(int e : adj[v]) {
        int u = edge[e].first ^ edge[e].second ^ v;
        if(!disc[u]) {
            compute(u, e);
            low[v] = min(low[v], low[u]);
            if(low[u] > disc[v]) bridge[e] = 1;
        } else if(e != p) {
            low[v] = min(low[v], disc[u]);
        }
    }
}

void addEdge(int u, int v) {
    adj[u].pb(sz(edge));
    adj[v].pb(sz(edge));
    edge.pb({u, v});
}

void findBridges(int n) {
    for(int i = 0; i < n; i++) {
        low[i] = disc[i] = 0;
    }
    for(int i = 0; i < sz(edge); i++) {
        bridge[i] = 0;
    }
    t_in = 0;
    for(int i = 0; i < n; i++) {
        if(!disc[i]) compute(i);
    }
}

// Must be call between testcases
void clearGraph(int n) {
    for(int i = 0; i < n; i++)
        adj[i].clear();
    edge.clear();
}
```

two_sat.cpp

```
/* Returns true if a given set of conditions of the form a v b is
 * satisfiable. NOTE: you must clear adj and adjt between testcases.
 *
 * n - (input) number of atomic propositions
 * assign - one valid assignation of truth values
 *
 * Complexity: O(n)
 */

// Useful when negation of atomic expression is given as a negative number
int conv(int x) {
    if(x > 0) return (x - 1) << 1;
    else return (-x - 1) << 1 | 1;
}

vector<int> adj[N], adjt[N];
```

```
void addEdge(int u, int v) {
    adj[u].pb(v);
    adjt[v].pb(u);
}

// Add the condition a v b
void addProp(int a, int b) {
    a = conv(a), b = conv(b);
    addEdge(a^1, b);
    addEdge(b^1, a);
}

bool vis[N];
int comp[N];
stack<int> order;

void toposort(int v) {
    vis[v] = 1;
    for(int u: adj[v]) if(!vis[u])
        toposort(u);
    order.push(v);
}

void mark(int v, int id) {
    comp[v] = id;
    for(int u: adjt[v]) if(comp[u] == -1)
        mark(u, id);
}

bool assign[N];

// Call this function after adding al conditions with addProp
bool twoSat(int n) {
    for(int i = 0; i < (n << 1); i++) {
        comp[i] = -1;
        vis[i] = 0;
    }
    for(int i = 0; i < (n << 1); i++) if(!vis[i])
        toposort(i);
    int cont = 0;
    for(int i = 0; i < (n << 1); i++) {
        int v = order.top();
        order.pop();
        if(comp[v] == -1)
            mark(v, cont++);
    }
    for(int i = 0; i < n; i++) {
        if(comp[i << 1] == comp[i << 1 | 1])
            return 0;
        assign[i] = comp[i << 1] > comp[i << 1 | 1];
    }
    return 1;
}
```

tarjan_scc.cpp

```
/* Computes the Strongly Connected Components of a graph
 *
 * n - (input) number of nodes of the graph
 * adj - (input) vector of adjacency of the graph
 * cont - number of SCCs of the graph
 * SCC[i] - list of nodes inside the i-th SCC
 * belong[i] - the index of the SCC in which the node i is in
 * adjSCC - vector of adjacency of the compressed graph
 *
 * Complexity: O(n) to compute cont, SCC and belong
 *             O(n log(n)) to compute adjSCC
 */

int n, temp, cont;
vector<int> adj[N], SCC[N], adjSCC[N];
bool vis[N];
int low[N], disc[N], belong[N];
stack<int> S;

void dfs(int v) {
    disc[v] = low[v] = ++temp;
    S.push(v);
    vis[v] = 1;
    for(int u: adj[v]) {
        if(!disc[u])
```

```
            dfs(u);
        if(vis[u])
            low[v] = min(low[u], low[v]);
    }
    if(disc[v] == low[v]) {
        while(1) {
            int u = S.top();
            S.pop();
            vis[u] = 0;
            belong[u] = cont;
            SCC[cont].pb(u);
            if(u == v) break;
        }
        cont++;
    }
}

void tarjan() {
    for(int i = 0; i < n; i++)
        if(!disc[i])
            dfs(i);
    for(int i = 0; i < cont; i++) {
        for(int u: SCC[i])
            for(int v: adj[u]) if(belong[v] != i) {
                adjSCC[i].pb(belong[v]);
            }
        sort(adjSCC[i].begin(), adjSCC[i].end());
        adjSCC[i].resize(unique(adjSCC[i].begin(), adjSCC[i].end()) - adjSCC[i].begin());
    }
}
```

Math

factorize.cpp, mod_exp.cpp, mod_inv_any.cpp, miller_rabin.cpp, extended_gcd.cpp, phi.cpp, divisors.cpp, mod_inv.cpp, polynomials_operations_full.cpp, gcd.cpp, polynomials_operations.cpp, gauss.cpp, inverse1n.cpp, gauss_mod.cpp, gauss_mod2.cpp, matrix_exp.cpp

factorize.cpp

```
/* Gets prime factors of a number
 *
 * a - positve integer
 * f - output vector for the factors
 *
 * Complexity: O(sqrt(n))
 */
void getFactors(ll n, vector<ll> &factors) {
    for (ll p = 2; p * p <= n; p++) {
        while (n % p == 0) {
            factors.push_back(p);
            n /= p;
        }
    }
    if (n > 1) factors.push_back(n);
}
```

mod_exp.cpp

```
/* Logarithmic modular exponentiation
 *
 * b - base
 * e - exponent
 * m - module
 *
 * Complexity: O(log(e))
 */
ll modexp(ll b, ll e, ll m = MOD) {
    ll res = 1;
    b %= m;
    while (e > 0) {
        if (e & 1) res = (res * b) % m;
        b = (b * b) % m;
        e /= 2;
    }
    return res;
}
```

mod_inv_any.cpp

```
/* Returns the modular multiplicative inverse of a number or -1 if gcd(a, m) != 1
 *
 * a, m - positive integers
 *
 * Complexity: O(log(m))
 */
ll gcdExt(ll a, ll b, ll &x, ll &y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    ll x1, y1;
    ll res = gcdExt(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    return res;
}

ll inv(ll a, ll m = MOD) {
    ll x, y;
    ll g = gcdExt(a, m, x, y);
    if (g != 1) return -1;
    return (m + x % m) % m;
}
```

miller_rabin.cpp

```
ll modexp(ll b, ll e, ll m = MOD) {
    ll res = 1;
    b %= m;
    while (e > 0) {
        if (e & 1) res = (res * b) % m;
        b = (b * b) % m;
        e /= 2;
    }
    return res;
}

/*
    Returns true if n is composite

    n - positive integer
    a - fermat base 2 <= a <= n - 2
    d, s - greatest s such that = 2^s * d
*/
bool checkComposite(ll n, ll a, ll d, int s) {
    ll x = modexp(a, d, n);
    if (x == 1 || x == n - 1) return false;
    for (int i = 1; i < s; i++) {
        x = (x * x) % n;
        if (x == n - 1) return false;
    }
    return true;
}

/* Returns true if a number is prime. Deterministic for 64-bit integers.
 *
 * n - positive integer
 *
 * Complexity: O(log(n)) the constant is at least 12
 */
bool millerRabin(ll n) {
    if (n < 2) return false;

    int r = 0;
    ll d = n - 1;
    while ((d & 1) == 0) {
        d >>= 1;
        r++;
    }

    for (int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {
        if (n == a) return true;
        if (checkComposite(n, a, d, r)) return false;
    }
    return true;
}
```

extended_gcd.cpp

```
/* Extended Euclidean algorithm. Returns gcd(a, b) and set the parameters
 * x and y to numbers such that ax + by = gcd(a, b).
 *
 * Time complexity: O(log(min(a, b)))
*/
ll gcdExt(ll a, ll b, ll &x, ll &y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    ll x1, y1;
    ll res = gcdExt(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    return res;
}
```

phi.cpp

```
/* Count how many numbers 1 to n are relatively prime with n
 *
 * n - positive integer
 *
 * Complexity: O(sqrt(n))
 */
ll phi(ll n) {
    ll res = n;
    for (ll p = 2; p * p <= n; p++) {
        if (n % p == 0) {
            while (n % p == 0) n /= p;
            res -= res / p;
        }
    }
    if (n > 1) res -= res / n;
    return res;
}
```

divisors.cpp

```
/* Get divisors of a number
 *
 * a - number
 * d - output param to store the divisors (needs to be empty)
 *
 * Complexity: O(n)
 */
void getDivisors(ll a, vector<ll> &d) {
    for (ll i = 1; i * i <= a; i++) {
        if (a % i == 0) {
            d.push_back(i);
            if (i * i != a) d.push_back(a / i);
        }
    }
}
```

mod_inv.cpp

```
/* Returns the modular multiplicative inverse of a number
 *
 * a - integer
 * m - prime module
 *
 * Complexity: O(log(m))
 */
ll modexp(ll b, ll e, ll m = MOD) {
    ll res = 1;
    b %= m;
    while (e > 0) {
        if (e & 1) res = (res * b) % m;
        b = (b * b) % m;
        e /= 2;
    }
    return res;
}

ll inv(ll a, ll m = MOD) {
    return modexp(a, m-2, m);
}
```

polynomials_operations_full.cpp

```
/* Useful polynomials operations of competitive programming
 * From https://github.com/e-maxx-eng/e-maxx-eng-aux/blob/master/src/polynomial.cpp
 *
 * Complexity: varies
 */

#include <bits/stdc++.h>
```

```
using namespace std;

namespace algebra {
    const int maxn = 1 << 21;
    const int magic = 250; // threshold for sizes to run the naive algo
    mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());

    template<typename T>
    T bpow(T x, size_t n) {
        if(n == 0) {
            return T(1);
        } else {
            auto t = bpow(x, n / 2);
            t *= t;
            return n % 2 ? x * t : t;
        }
    }

    template<int m>
    struct modular {

        // https://en.wikipedia.org/wiki/Berlekamp-Rabin_algorithm
        // solves x^2 = y (mod m) assuming m is prime in O(log m).
        // returns nullopt if no sol.
        optional<modular> sqrt() const {
            static modular y;
            y = *this;
            if(r == 0) {
                return 0;
            } else if(bpow(y, (m - 1) / 2) != modular(1)) {
                return nullopt;
            } else {
                while(true) {
                    modular z = rng();
                    if(z * z == *this) {
                        return z;
                    }
                    struct lin {
                        modular a, b;
                        lin(modular a, modular b): a(a), b(b) {}
                        lin(modular a): a(a), b(0) {}
                        lin operator * (const lin& t) {
                            return {
                                a * t.a + b * t.b * y,
                                a * t.b + b * t.a
                            };
                        }
                    } x(z, 1); // z + x
                    x = bpow(x, (m - 1) / 2);
                    if(x.b != modular(0)) {
                        return x.b.inv();
                    }
                }
            }
        }

        int64_t r;
        modular() : r(0) {}
        modular(int64_t rr) : r(rr) {if(abs(r) >= m) r %= m; if(r < 0) r += m;}
        modular inv() const {return bpow(*this, m - 2);}
        modular operator - () const {return r ? m - r : 0;}
        modular operator * (const modular &t) const {return r * t.r % m;}
        modular operator / (const modular &t) const {return *this * t.inv();}
        modular operator += (const modular &t) {r += t.r; if(r >= m) r -= m; return *this;}
        modular operator -= (const modular &t) {r -= t.r; if(r < 0) r += m; return *this;}
        modular operator + (const modular &t) const {return modular(*this) += t;}
        modular operator - (const modular &t) const {return modular(*this) -= t;}
        modular operator *= (const modular &t) {return *this = *this * t;}
        modular operator /= (const modular &t) {return *this = *this / t;}

        bool operator == (const modular &t) const {return r == t.r;}
        bool operator != (const modular &t) const {return r != t.r;}

        operator int() const {return r;}
        int64_t rem() const {return 2 * r > m ? r - m : r;}
    };

    template<int T>
    istream& operator >> (istream &in, modular<T> &x) {
        return in >> x.r;
    }

    template<typename T>
    T fact(int n) {
        static T F[maxn];
        return F[n] ? F[n] : F[n] = n ? fact<T>(n - 1) * T(n) : T(1);
    }

    template<typename T>
    T rfact(int n) {
        static T RF[maxn];
        return RF[n] ? RF[n] : RF[n] = T(1) / fact<T>(n);
    }

    namespace fft {
        typedef double ftype;
        typedef complex<ftype> point;

        point w[maxn];
        const ftype pi = acos(-1);
        bool initiated = 0;
        void init() {
            if(!initiated) {
                for(int i = 1; i < maxn; i *= 2) {
                    for(int j = 0; j < i; j++) {
                        w[i + j] = polar(ftype(1), pi * j / i);
                    }
                }
                initiated = 1;
            }
        }

        void fft(auto *in, point *out, int n, int k = 1) {
            if(n == 1) {
                *out = *in;
            } else {
                n /= 2;
                fft(in, out, n, 2 * k);
                fft(in + k, out + n, n, 2 * k);
                for(int i = 0; i < n; i++) {
                    auto t = out[i + n] * w[i + n];
                    out[i + n] = out[i] - t;
                    out[i] += t;
                }
            }
        }

        void mul_slow(vector<auto> &a, const vector<auto> &b) {
            if(a.empty() || b.empty()) {
                a.clear();
            } else {
                a.resize(a.size() + b.size() - 1);
                for(int k = a.size() - 1; k >= 0; k--) {
                    a[k] *= b[0];
                    for(int j = 1; j < min(k + 1, (int)b.size()); j++) {
                        a[k] += a[k - j] * b[j];
                    }
                }
            }
        }

        template<typename T>
        void mul(vector<T> &a, vector<T> b) {
            if(min(a.size(), b.size()) < magic) {
                mul_slow(a, b);
                return;
            }
            init();
            static const T split = 1 << 15;
            size_t n = a.size() + b.size() - 1;
            while(__builtin_popcount(n) != 1) {
                n++;
            }
            a.resize(n);
            b.resize(n);
            static point A[maxn], B[maxn];
            static point C[maxn], D[maxn];
            for(size_t i = 0; i < n; i++) {
                A[i] = point(a[i].rem() % split, a[i].rem() / split);
                B[i] = point(b[i].rem() % split, b[i].rem() / split);
            }
            fft(A, C, n); fft(B, D, n);
            for(size_t i = 0; i < n; i++) {
                A[i] = C[i] * (D[i] + conj(D[(n - i) % n]));
                B[i] = C[i] * (D[i] - conj(D[(n - i) % n]));
            }
            fft(A, C, n); fft(B, D, n);
            reverse(C + 1, C + n);
            reverse(D + 1, D + n);
            int t = 2 * n;
            for(size_t i = 0; i < n; i++) {
                T A0 = llround(real(C[i]) / t);
                T A1 = llround(imag(C[i]) / t + imag(D[i]) / t);
                T A2 = llround(real(D[i]) / t);
                a[i] = A0 + A1 * split - A2 * split * split;
            }
        }
    }

    template<typename T>
    struct poly {
        vector<T> a;

        void normalize() { // get rid of leading zeroes
            while(!a.empty() && a.back() == T(0)) {
                a.pop_back();
            }
        }

        poly(){}
        poly(T a0) : a{a0}{normalize();}
        poly(const vector<T> &t) : a(t){normalize();}

        poly operator -() const {
            auto t = *this;
            for(auto &it: t.a) {
                it = -it;
            }
            return t;
        }

        poly operator += (const poly &t) {
            a.resize(max(a.size(), t.a.size()));
            for(size_t i = 0; i < t.a.size(); i++) {
                a[i] += t.a[i];
            }
            normalize();
            return *this;
        }

        poly operator -= (const poly &t) {
            a.resize(max(a.size(), t.a.size()));
            for(size_t i = 0; i < t.a.size(); i++) {
                a[i] -= t.a[i];
            }
            normalize();
            return *this;
        }
        poly operator + (const poly &t) const {return poly(*this) += t;}
        poly operator - (const poly &t) const {return poly(*this) -= t;}

        poly mod_xk(size_t k) const { // get first k coefficients
            return vector<T>(begin(a), begin(a) + min(k, a.size()));
        }

        poly mul_xk(size_t k) const { // multiply by x^k
            auto res = a;
            res.insert(begin(res), k, 0);
            return res;
        }

        poly div_xk(size_t k) const { // drop first k coefficients
            return vector<T>(begin(a) + min(k, a.size()), end(a));
        }

        poly substr(size_t l, size_t r) const { // return mod_xk(r).div_xk(l)
            return vector<T>(
                begin(a) + min(l, a.size()),
                begin(a) + min(r, a.size())
            );
        }

        poly inv(size_t n) const { // get inverse series mod x^n
            assert((*this)[0] != T(0));
            poly ans = T(1) / a[0];
            size_t a = 1;
            while(a < n) {
                poly C = (ans * mod_xk(2 * a)).substr(a, 2 * a);
                ans -= (ans * C).mod_xk(a).mul_xk(a);
                a *= 2;
            }
            return ans.mod_xk(n);
        }
        poly operator *= (const poly &t) {fft::mul(a, t.a); normalize(); return *this;}
        poly operator * (const poly &t) const {return poly(*this) *= t;}

        poly reverse(size_t n) const { // computes x^n A(x^{-1})
            auto res = a;
            res.resize(max(n, res.size()));
            return vector<T>(res.rbegin(), res.rbegin() + n);
        }

        poly reverse() const {
            return reverse(deg() + 1);
        }

        pair<poly, poly> divmod_slow(const poly &b) const { // when divisor or quotient is small
            vector<T> A(a);
            vector<T> res;
            while(A.size() >= b.a.size()) {
                res.push_back(A.back() / b.a.back());
                if(res.back() != T(0)) {
                    for(size_t i = 0; i < b.a.size(); i++) {
                        A[A.size() - i - 1] -= res.back() * b.a[b.a.size() - i - 1];
                    }
                }
                A.pop_back();
            }
            std::reverse(begin(res), end(res));
            return {res, A};
        }

        pair<poly, poly> divmod(const poly &b) const { // returns quotiend and remainder of a mod b
            assert(!b.is_zero());
            if(deg() < b.deg()) {
                return {poly{0}, *this};
            }
            int d = deg() - b.deg();
            if(min(d, b.deg()) < magic) {
                return divmod_slow(b);
            }
            poly D = (reverse().mod_xk(d + 1) * b.reverse().inv(d + 1)).mod_xk(d + 1).reverse(d +
1);
            return {D, *this - D * b};
        }

        poly operator / (const poly &t) const {return divmod(t).first;}
        poly operator % (const poly &t) const {return divmod(t).second;}
        poly operator /= (const poly &t) {return *this = divmod(t).first;}
        poly operator %= (const poly &t) {return *this = divmod(t).second;}
        poly operator *= (const T &x) {
            for(auto &it: a) {
                it *= x;
            }
            normalize();
            return *this;
        }
        poly operator /= (const T &x) {
            for(auto &it: a) {
                it /= x;
            }
            normalize();
            return *this;
        }
        poly operator * (const T &x) const {return poly(*this) *= x;}
        poly operator / (const T &x) const {return poly(*this) /= x;}

        poly conj() const { // A(x) -> A(-x)
            auto res = *this;
            for(int i = 1; i <= deg(); i += 2) {
                res[i] = -res[i];
            }
            return res;
        }

        void print(int n) const {
            for(int i = 0; i < n; i++) {
                cout << (*this)[i] << ' ';
            }
            cout << "\n";
        }

        void print() const {
            print(deg() + 1);
        }

        T eval(T x) const { // evaluates in single point x
            T res(0);
            for(int i = deg(); i >= 0; i--) {
                res *= x;
                res += a[i];
            }
            return res;
        }

        T lead() const { // leading coefficient
            assert(!is_zero());
            return a.back();
        }

        int deg() const { // degree, -1 for P(x) = 0
            return (int)a.size() - 1;
        }

        bool is_zero() const {
            return a.empty();
        }

        T operator [](int idx) const {
            return idx < 0 || idx > deg() ? T(0) : a[idx];
        }

        T& coef(size_t idx) { // mutable reference at coefficient
            return a[idx];
        }

        bool operator == (const poly &t) const {return a == t.a;}
        bool operator != (const poly &t) const {return a != t.a;}

        poly deriv() { // calculate derivative
            vector<T> res(deg());
            for(int i = 1; i <= deg(); i++) {
                res[i - 1] = T(i) * a[i];
            }
            return res;
        }

        poly integr() { // calculate integral with C = 0
            vector<T> res(deg() + 2);
            for(int i = 0; i <= deg(); i++) {
                res[i + 1] = a[i] / T(i + 1);
            }
            return res;
        }

        size_t trailing_xk() const { // Let p(x) = x^k * t(x), return k
            if(is_zero()) {
                return -1;
            }
            int res = 0;
            while(a[res] == T(0)) {
                res++;
            }
            return res;
        }

        poly log(size_t n) { // calculate log p(x) mod x^n
            assert(a[0] == T(1));
            return (deriv().mod_xk(n) * inv(n)).integr().mod_xk(n);
        }

        poly exp(size_t n) { // calculate exp p(x) mod x^n
            if(is_zero()) {
                return T(1);
            }
            assert(a[0] == T(0));
            poly ans = T(1);
            size_t a = 1;
            while(a < n) {
                poly C = ans.log(2 * a).div_xk(a) - substr(a, 2 * a);
                ans -= (ans * C).mod_xk(a).mul_xk(a);
                a *= 2;
            }
            return ans.mod_xk(n);
        }

        poly pow_bin(int64_t k, size_t n) { // O(n log n log k)
            if(k == 0) {
                return poly(1).mod_xk(n);
            } else {
                auto t = pow(k / 2, n);
                t *= t;
                return (k % 2 ? *this * t : t).mod_xk(n);
            }
        }

        // O(d * n) with the derivative trick from
        // https://codeforces.com/blog/entry/73947?#comment-581173
        poly pow_dn(int64_t k, size_t n) {
            vector<T> Q(n);
            Q[0] = bpow(a[0], k);
            for(int i = 1; i < (int)n; i++) {
                for(int j = 0; j <= min(deg(), i + 1); j++) {
                    Q[i] += a[j] * Q[i - j] * (T(k + 1) * T(j) - T(i));
                }
                Q[i + 1] /= i * a[0];
            }
            return Q;
        }

        // calculate p^k(n) mod x^n in O(n log n)
        // might be quite slow due to high constant
        poly pow(int64_t k, size_t n) {
            if(is_zero()) {
                return *this;
            }
            int i = trailing_xk();
            if(i > 0) {
                return i * k >= (int64_t)n ? poly(0) : div_xk(i).pow(k, n - i * k).mul_xk(i * k);
            }
            if(min(deg(), (int)n) <= magic) {
                return pow_dn(k, n);
            }
            if(k <= magic) {
                return pow_bin(k, n);
            }
            T j = a[i];
            poly t = *this / j;
            return bpow(j, k) * (t.log(n) * T(k)).exp(n).mod_xk(n);
        }

        // returns nullopt if undefined
        optional<poly> sqrt(size_t n) const {
            if(is_zero()) {
                return *this;
            }
            int i = trailing_xk();
            if(i % 2) {
                return nullopt;
            } else if(i > 0) {
                auto ans = div_xk(i).sqrt(n - i / 2);
                return ans ? ans->mul_xk(i / 2) : ans;
            }
            auto st = (*this)[0].sqrt();
            if(st) {
                poly ans = *st;
                size_t a = 1;
                while(a < n) {
                    a *= 2;
                    ans -= (ans - mod_xk(a) * ans.inv(a)).mod_xk(a) / 2;
                }
                return ans.mod_xk(n);
            }
            return nullopt;
        }

        poly mulx(T a) { // component-wise multiplication with a^k
            T cur = 1;
            poly res(*this);
            for(int i = 0; i <= deg(); i++) {
                res.coef(i) *= cur;
                cur *= a;
            }
            return res;
        }

        poly mulx_sq(T a) { // component-wise multiplication with a^{k^2}
            T cur = a;
            T total = 1;
            T aa = a * a;
            poly res(*this);
            for(int i = 0; i <= deg(); i++) {
                res.coef(i) *= total;
                total *= cur;
                cur *= aa;
            }
            return res;
        }
        vector<T> chirpz_even(T z, int n) { // P(1), P(z^2), P(z^4), ..., P(z^2(n-1))
            int m = deg();
            if(is_zero()) {
                return vector<T>(n, 0);
            }
            vector<T> vv(m + n);
            T zi = T(1) / z;
            T zz = zi * zi;
            T cur = zi;
            T total = 1;
            for(int i = 0; i <= max(n - 1, m); i++) {
                if(i <= m) {vv[m - i] = total;}
                if(i < n) {vv[m + i] = total;}
                total *= cur;
                cur *= zz;
            }
            poly w = (mulx_sq(z) * vv).substr(m, m + n).mulx_sq(z);
            vector<T> res(n);
            for(int i = 0; i < n; i++) {
                res[i] = w[i];
            }
            return res;
        }

        vector<T> chirpz(T z, int n) { // P(1), P(z), P(z^2), ..., P(z^(n-1))
            auto even = chirpz_even(z, (n + 1) / 2);
            auto odd = mulx(z).chirpz_even(z, n / 2);
            vector<T> ans(n);
            for(int i = 0; i < n / 2; i++) {
                ans[2 * i] = even[i];
                ans[2 * i + 1] = odd[i];
            }
            if(n % 2 == 1) {
                ans[n - 1] = even.back();
            }
            return ans;
        }

        vector<T> eval(vector<poly> &tree, int v, auto l, auto r) { // auxiliary evaluation function
            if(r - l == 1) {
                return {eval(*l)};
            } else {
                auto m = l + (r - l) / 2;
                auto A = (*this % tree[2 * v]).eval(tree, 2 * v, l, m);
                auto B = (*this % tree[2 * v + 1]).eval(tree, 2 * v + 1, m, r);
                A.insert(end(A), begin(B), end(B));
                return A;
            }
        }

        vector<T> eval(vector<T> x) { // evaluate polynomial in (x1, ..., xn)
            int n = x.size();
            if(is_zero()) {
                return vector<T>(n, T(0));
            }
            vector<poly> tree(4 * n);
            build(tree, 1, begin(x), end(x));
            return eval(tree, 1, begin(x), end(x));
        }

        poly inter(vector<poly> &tree, int v, auto l, auto r, auto ly, auto ry) { // auxiliary
interpolation function
            if(r - l == 1) {
                return {*ly / a[0]};
            } else {
                auto m = l + (r - l) / 2;
                auto my = ly + (ry - ly) / 2;
                auto A = (*this % tree[2 * v]).inter(tree, 2 * v, l, m, ly, my);
                auto B = (*this % tree[2 * v + 1]).inter(tree, 2 * v + 1, m, r, my, ry);
                return A * tree[2 * v + 1] + B * tree[2 * v];
            }
        }

        static auto resultant(poly a, poly b) { // computes resultant of a and b
            if(b.is_zero()) {
                return 0;
            } else if(b.deg() == 0) {
                return bpow(b.lead(), a.deg());
            } else {
                int pw = a.deg();
                a %= b;
                pw -= a.deg();
                auto mul = bpow(b.lead(), pw) * T((b.deg() & a.deg() & 1) ? -1 : 1);
                auto ans = resultant(b, a);
                return ans * mul;
            }
        }

        static poly kmul(auto L, auto R) { // computes (x-a1)(x-a2)...(x-an) without building tree
            if(R - L == 1) {
                return vector<T>{-*L, 1};
            } else {
                auto M = L + (R - L) / 2;
                return kmul(L, M) * kmul(M, R);
            }
        }

        static poly build(vector<poly> &res, int v, auto L, auto R) { // builds evaluation tree for
(x-a1)(x-a2)...(x-an)
            if(R - L == 1) {
                return res[v] = vector<T>{-*L, 1};
            } else {
                auto M = L + (R - L) / 2;
                return res[v] = build(res, 2 * v, L, M) * build(res, 2 * v + 1, M, R);
            }
        }

        static auto inter(vector<T> x, vector<T> y) { // interpolates minimum polynomial from (xi,
yi) pairs
            int n = x.size();
            vector<poly> tree(4 * n);
            return build(tree, 1, begin(x), end(x)).deriv().inter(tree, 1, begin(x), end(x),
begin(y), end(y));
        }


        static poly xk(size_t n) { // P(x) = x^n
            return poly(T(1)).mul_xk(n);
        }

        static poly ones(size_t n) { // P(x) = 1 + x + ... + x^{n-1}
            return vector<T>(n, 1);
        }

        static poly expx(size_t n) { // P(x) = e^x (mod x^n)
            return ones(n).borel();
        }

        // [x^k] (a corr b) = sum_{i+j=k} ai*b{m-j}
        //                  = sum_{i-j=k-m} ai*bj
        static poly corr(poly a, poly b) { // cross-correlation
            return a * b.reverse();
        }

        poly invborel() const { // ak *= k!
            auto res = *this;
            for(int i = 0; i <= deg(); i++) {
                res.coef(i) *= fact<T>(i);
            }
            return res;
        }

        poly borel() const { // ak /= k!
            auto res = *this;
            for(int i = 0; i <= deg(); i++) {
                res.coef(i) *= rfact<T>(i);
            }
            return res;
        }

        poly shift(T a) const { // P(x + a)
            return (expx(deg() + 1).mulx(a).reverse() * invborel()).div_xk(deg()).borel();
        }
    };

    static auto operator * (const auto& a, const poly<auto>& b) {
        return b * a;
    }
};

using namespace algebra;

const int mod = 998244353;
typedef modular<mod> base;
typedef poly<base> polyn;

void solve() {
    int n, m;
    cin >> n >> m;
    vector<base> a(n);
    copy_n(istream_iterator<base>(cin), n, begin(a));
    polyn(a).pow(m, n).print(n);
}

signed main() {
    //freopen("input.txt", "r", stdin);
    ios::sync_with_stdio(0);
    cin.tie(0);
    int t;
    t = 1;// cin >> t;
    while(t--) {
        solve();
    }
}
```

gcd.cpp

```
/* Greatest common divisor
 *
 * a, b - non-negative integers
 *
 * Complexity: O(log(min(a, b)))
 */
ll gcd(ll a, ll b) {
    return b ? gcd(b, a % b) : a;
}
```

polynomials_operations.cpp

```
/* Useful polynomials operation of competitive programming
 * From https://github.com/e-maxx-eng/e-maxx-eng-aux/blob/master/src/polynomial.cpp
 * The main operations are kept in this file.
 *
 * Complexity: varies
 */
struct Poly {
    vector<ll> p;

    Poly() {};

    Poly(int deg) { // for deg < 0, p(x) = 0
        p.resize(max(0, deg+1));
    }

    Poly(vector<ll> &coefs) : p(coefs) {
        cout << coefs[0] << " " << coefs[1] << endl;
    }

    int deg() { // degree is -1 for p(x) = 0
        return int(p.size()) - 1;
    }

    Poly operator*(ll x) { // O(deg())
        Poly b = *this;
        for (ll &c : b.p) c *= x;
        return b;
    }

    Poly operator*=(ll x) {
        return *this = *this * x;
    }

    Poly mul_slow(Poly &a) { // O(deg() * a.deg())
        Poly b(deg() + a.deg());
        for (int i = 0; i <= deg(); i++) {
            for (int j = 0; j <= a.deg(); j++) {
                b.p[i+j] += p[i] * a.p[j];
            }
        }
        return b;
    }

    void print() { // debug
        for (int i = deg(); i >= 0; i--) {
            if (p[i] < 0) cout << "- ";
            else cout << "+ ";
            cout << abs(p[i]);
            if (i != 0) cout << "x^" << i << " ";
        }
        cout << endl;
    }
};
```

gauss.cpp

```
/* Solves a system of linear equations
 *
 * Complexity: O(n^3)
 */
struct Gauss {
  int n, m;
  vector<int> pos;
  int rank = 0;
  vector<vector<double>> a;
  const double EPS = 1e-9;

  // n equations, m-1 variables, last column is for coefficients
  Gauss(int n, int m, vector<vector<double>> &a) : n(n), m(m), a(a) {
    pos.assign(m, -1);
  }

  /* if a solution exists, it will be stored in ans
    0 - no solution
    1 - unique solution
    2 - infinite number of solutions */
  int solve(vector<double> &ans) {
    for (int col = 0, row = 0; col < m && row < n; col++) {
      int sel = row;
      for (int i = row+1; i < n; i++) {
        if (abs(a[i][col]) > abs(a[sel][col])) sel = i;
      }

      if (abs(a[sel][col]) < EPS) continue;

      swap(a[sel], a[row]);

      pos[col] = row;

      for (int i = 0; i < n; i++) {
        if (i != row) {
          double mult = a[i][col] / a[row][col];
          a[i][col] = 0.0;
          for (int j = col+1; j <= m; j++) {
            a[i][j] -= a[row][j] * mult;
          }
        }
      }

      ++row, ++rank;
    }

    ans.assign(m, 0);

    bool multiple = false;

    for (int i = 0; i < m; i++) {
      if (pos[i] != -1) ans[i] = a[pos[i]][m] / a[pos[i]][i];
      else multiple = true;
    }

    for (int i = 0; i < n; i++) {
      double sum = 0.0;
      for (int j = 0; j < m; j++) {
        sum += ans[j] * a[i][j];
      }
      if (abs(sum - a[i][m]) > EPS) return 0;
    }

    if (multiple) return 2;

    return 1;
  }
};
```

inverse1n.cpp

```
/* Modular inverse for numbers 1 to n mod m.
 *
 * Complexity: O(n)
 */
void inverse_1n(int n, int m) {
    vector<int> inv(n+1);
    inv[1] = 1;
    for (int i = 2; i <= n; i++) {
        inv[i] = m - ((m/i) * inv[m%i]) % m;
    }

}
```

gauss_mod.cpp

```
/* Solves a system of linear equations in Z_mod
 *
 * Complexity: O(n^3)
 */
ll gcdExt(ll a, ll b, ll &x, ll &y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    ll x1, y1;
    ll res = gcdExt(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    return res;
}

ll inv(ll a, ll m = MOD) {
    ll x, y;
    ll g = gcdExt(a, m, x, y);
    if (g != 1) return -1;
    return (m + x % m) % m;
}

struct Gauss {
  int n, m;
  vector<int> pos;
  int rank = 0;
  ll mod;
  vector<vector<ll>> a;

  // n equations, m-1 variables, last column is for coefficients
  Gauss(int n, int m, ll mod, vector<vector<ll>> &a) : n(n), m(m), mod(mod), a(a) {
    pos.assign(m, -1);
  }

  /* if a solution exists, it will be stored in ans
    0 - no solution
    1 - unique solution
    2 - infinite number of solutions */
  int solve(vector<ll> &ans) {
    for (int col = 0, row = 0; col < m && row < n; col++) {
      int sel = row;
      for (int i = row+1; i < n; i++) {
        if (a[i][col] > 0) {
          sel = i;
          break;
        }
      }

      if (a[sel][col] == 0) continue;

      swap(a[sel], a[row]);

      pos[col] = row;

      for (int i = 0; i < n; i++) {
        if (i != row) {
          ll mult = a[i][col] * inv(a[row][col], mod) % mod;
          a[i][col] = 0;
          for (int j = col+1; j <= m; j++) {
            a[i][j] = (a[i][j] + mod - a[row][j] * mult % mod) % mod;
          }
        }
      }

      ++row, ++rank;
    }

    ans.assign(m, 0);

    bool multiple = false;

    for (int i = 0; i < m; i++) {
      if (pos[i] != -1) ans[i] = a[pos[i]][m] * inv(a[pos[i]][i], mod) % mod;
      else multiple = true;
    }

    for (int i = 0; i < n; i++) {
      ll sum = 0.0;
      for (int j = 0; j < m; j++) {
        sum = (sum + ans[j] * a[i][j] % mod) % mod;
      }
      if (sum != a[i][m]) return 0;
    }

    if (multiple) return 2;

    return 1;
  }
};
```

gauss_mod2.cpp

```
/* Solves a system of linear equations in Z_2, it is faster than gauss_mod
 * by the use of bitset.
 *
 * Complexity: O(n^3)
 */
template<int M>
struct Gauss {
  int n, m;
  array<int, M> pos;
  int rank = 0;
  vector<bitset<M>> a;

  // n equations, m-1 variables, last column is for coefficients
  Gauss(int n, int m, vector<bitset<M>> &a) : n(n), m(m), a(a) {
    pos.fill(-1);
  }

  int solve(bitset<N> &ans) {
    for (int col = 0, row = 0; col < m && row < n; col++) {
      int one = -1;
      for (int i = row; i < n; i++) {
        if (a[i][col]) {
          one = i;
          break;
        }
      }

      if (one == -1) { continue; }

      swap(a[one], a[row]);

      pos[col] = row;

      for (int i = row + 1; i < n; i++) {
        if (a[i][col])
          a[i] ^= a[row];
      }

      ++row, ++rank;
    }

    ans.reset();

    for (int i = m - 1; i >= 0; i--) {
      if (pos[i] == -1) ans[i] = true;
      else {
        int k = pos[i];
        for (int j = i + 1; j < m; j++) if (a[k][j]) ans[i] = ans[i] ^ ans[j];
        ans[i] = ans[i] ^ a[k][m];
      }
    }

    for (int i = rank; i < n; i++) if (a[i][m]) return 0;

    return 1;
  }
};
```

matrix_exp.cpp

```
/* Returns the matrix exponentiation
 *
 * base - d x d matrix
 * n - non-negative integer for the exponent
 * res - output param
 *
 * Complexity: O(d^2 * log(n))
 */
void matProd(vector<vector<ll>> &a, vector<vector<ll>> &b, vector<vector<ll>> &c) {
    int n = a.size();
    int m = a[0].size();
    int p = b[0].size();
    vector<vector<ll>> res(n, vector<ll>(p, 0));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < p; j++) {
            for (int k = 0; k < m; k++) {
                res[i][j] += a[i][k] * b[k][j];
            }
        }
    }
    c = res;
}

void matExp(vector<vector<ll>> &base, ll n, vector<vector<ll>> &res) {
    int d = base.size();
    for (int i = 0; i < d; i++) {
        for (int j = 0; j < d; j++) {
            res[i][j] = ll(i == j);
        }
    }
    while (n > 0) {
        if (n % 2 == 1) matProd(res, base, res);
        matProd(base, base, base);
        n /= 2;
    }
}
```

Numerical methods

`roots_newton.cpp, ternary_search.cpp, fast_double_bs.cpp, stable_sum.cpp, simpson_integration.cpp`

roots_newton.cpp

```
/* Find some root of a function f with derivative df.
 *
 * Complexity: The precision doubles for each iteration, in ideal conditions.
 * For roots with multiplicity greater than one, the precision increases linearly.
 */
double f(double x);
double df(double x);

double findRoot(double x0=0.0) {
    double x = x0;
    double fx = f(x);
    while (abs(fx) > EPS) {
        x -= fx / df(x);
        fx = f(x);
    }
    return x;
}
```

ternary_search.cpp

```
/* Find the minimum of a function that first strictly decreases, then has its
 * minimum and finally strictly increases
 *
 * Complexity: O(log(n))
 */

double ternary_search(double l, double r) {
    while (r - l > EPS) { // TODO set EPS
        double m1 = l + (r - l) / 3;
        double m2 = r - (r - l) / 3;
        if (f(m1) > f(m2)) l = m1; // < for maximum
        else r = m2;
    }
    return l;
}
```

fast_double_bs.cpp

```
/* Binary search with doubles considering that small numbers are
 * closer to each other, works only if l > 0
 */
double bs(double l, double r) {
    while (r - l > EPS) {
        double m;
        if (r > 2*l) m = sqrt(a*b);
        else m = (a + b) / 2;
    }
}
```

stable_sum.cpp

```
/* From Handbook of geometry for competitive programmers - Victor Lecomte
 * This is an algorithm to make sums of positive numbers more precise
 * Complexity is O(n) amortized and the relative error of the sum is
 * 2 log_2(n) eps, down from n * eps precision of the serial sum
 * eps is the machine precision.
 */

struct stableSum {
    int cnt = 0;
    vector<double> v, pref{0};
    // add number a to the sum
    void operator+=(double a) {
        assert(a >= 0);
        int s = ++cnt;
        while (s % 2 == 0) {
            a += v.back();
            v.pop_back();
            pref.pop_back();
            s /= 2;
        }
        v.push_back(a);
        pref.push_back(pref.back() + 1);
    }

    // return the sum value
    double val() {
        return pref.back();
    }
}
```

simpson_integration.cpp

```
/* Returns the integral of some function f in the interval [a, b].
 *
 * Complexity: O(2n)
 */
double f(double x);

double integral(double a, double b, int n) {
    n *= 2;
    double h = (b - a) / n;
    double s = f(a) + f(b);
    for (int i = 1; i < n; i++) {
        s += f(a + h*i) * (1 + (i & 1)) * 2;
    }
    return s * h / 3;
}
```

Strings

z_function.cpp, manacher.cpp, rolling_hash.cpp, kmp.cpp, suffix_automata.cpp, kmp_automata.cpp, suffix_array.cpp, min_cyclic_string.cpp

z_function.cpp

```
/* Computes z function, where z[i] is the length of the longest prefix of s[i, n)
 * that is also a prefix of s[0, n). For convention, this template assumes z[0] = 0
 *
 * Complexity: O(n)
 */

vector<int> z_function(string s) {
    int n = sz(s);
    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;
}
```

manacher.cpp

```
/* Computes the following for every position of the string s:
 *
 * best[0][i] - length of the longest palindrome that ends in s[i]
 * best[1][i] - length of the longest palindrome that starts in s[i]
 * d1[i] - number of odd length palindromes with center in s[i]
 * d2[i] - number of even length palindromes with center in s[i]
 *         (we consider the center of an even length palindrome as
 *         the rightmost of the two characters in the center)
 *
 * Complexity: O(n)
 */

int best[2][N];
int d1[N], d2[N];

void manacher(string &s){
    n = sz(s);
    for(int i = 0; i < n; i++)
        best[0][i] = best[1][i] = 1;
    for (int i = 0, l = 0, r = -1; i < n; i++) {
        int k = (i > r) ? 1 : min(d1[l + r - i], r - i + 1);
        while (0 <= i - k && i + k < n && s[i - k] == s[i + k]) {
            best[1][i-k] = max(best[1][i-k], k << 1 | 1);
            best[0][i+k] = max(best[0][i+k], k << 1 | 1);
            k++;
        }
        d1[i] = k--;
        if (i + k > r) {
            l = i - k;
            r = i + k;
        }
    }
    for (int i = 0, l = 0, r = -1; i < n; i++) {
        int k = (i > r) ? 0 : min(d2[l + r - i + 1], r - i + 1);
        while (0 <= i - k - 1 && i + k < n && s[i - k - 1] == s[i + k]) {
            best[1][i-k-1] = max(best[1][i-k-1], k*2 + 2);
            best[0][i+k] = max(best[0][i+k], k*2 + 2);
            k++;
        }
        d2[i] = k--;
        if (i + k > r) {
            l = i - k - 1;
            r = i + k ;
        }
    }
}
```

rolling_hash.cpp

```
/* Computes the suffixes of a string s. Can answer queries of the
 * any substring of s.
 *
 * START - first character of the alphabet
 * K - number of prime bases
 * P - array of prime bases
 *
 * Complexity: O(n) precomputation, O(1) queries
 */

const ll MOD = 1e9 + 7;
const ll K = 2;
const ll P[K] = {29, 31};
const char START = 'a';

array<ll, K> pot[N];

ll mul(ll a, ll b) {
    return a * b % MOD;
}

ll sum(ll a, ll b) {
    a += b;
    if(a >= MOD) a -= MOD;
    return a;
}

// Must be called once in the beginning of the program to compute
// powers of the prime bases
void prec() {
    for(int k = 0; k < K; k++)
        pot[0][k] = 1;
    for(int i = 1; i < N; i++)
        for(int k = 0; k < K; k++)
            pot[i][k] = mul(pot[i-1][k], P[k]);
}

struct hsh {
    vector<array<ll, K> > suf;
    hsh() {}
    hsh(string &s) : suf(sz(s) + 1) {
        for(int k = 0; k < K; k++) suf[sz(s)][k] = 0;
        for(int i = sz(s) - 1; i >= 0; i--)
            for(int k = 0; k < K; k++)
                suf[i][k] = sum(mul(suf[i + 1][k], P[k]), s[i] - START + 1);
    }
    // Queries the hashing of the substring s[l, r)
    array<ll, K> que(int l, int r) {
        array<ll, K> cur;
        for(int k = 0; k < K; k++)
            cur[k] = sum(suf[l][k], MOD - mul(suf[r][k], pot[r - l][k]));
        return cur;
    }
};
```

kmp.cpp

```
/* Computes pi, where pi[i] is the length of the longest (proper) prefix of s[0, i]
 * that is also a (proper) suffix of s[0, i]
 *
 * Complexity: O(n)
 */

int pi[N];

void kmp(string& s) {
    int n = sz(s), k = 0;
    for(int i = 1; i < n; i++){
        while(k && s[i] != s[k])
            k = pi[k-1];
        if(s[i] == s[k]) k++;
        pi[i] = k;
    }
}
```

suffix_automata.cpp

```
/* Online algorithm that computes the suffix automata of a string.
 * Every state of the suffix automata represents a set of end positions of some substrings
 * and every path on the suffix automata represents a different substring.
 * You can use the extend method to add character by character to the suffix automata
 * or use the constructor with a string to build the suffix automara for the whole
 * string at once.
 *
 * Complexity: O(n log ALPH) time for building the suffix automata for n characters.
 *             O(n) space
 */

struct suffixAutomata {
    struct state {
        int len, link;
        map<char, int> next;
        state() : len(0), link(-1) {}
        state(int len) : len(len) {}
        state(int len, int link, map<char, int> next)
            : len(len), link(link), next(next) {}
    };
    vector<state> st;
    int last;
    suffixAutomata() : last(0) {
        st.pb({});
    }
    suffixAutomata(string &s) : last(0) {
        st.pb({});
        for(auto c: s)
            extend(c);
    }
    void extend(char c) {
        st.pb({st[last].len + 1});
        int cur = sz(st) - 1;
        int p = last;
        while(p != -1 && !st[p].next.count(c)) {
            st[p].next[c] = cur;
            p = st[p].link;
        }
        if(p == -1) st[cur].link = 0;
        else {
            int q = st[p].next[c];
            if(st[p].len + 1 == st[q].len) st[cur].link = q;
            else {
                int clone = sz(st);
                st.pb({st[p].len + 1, st[q].link, st[q].next});
                while(p != -1 && st[p].next[c] == q) {
                    st[p].next[c] = clone;
                    p = st[p].link;
                }
                st[q].link = st[cur].link = clone;
            }
        }
        last = cur;
    }
};
```

kmp_automata.cpp

```
/* Computes the kmp automata. The i-th state of this automata corresponds to any string
 * which its longest suffix that is a prefix of string has length i. The entry kmp[i][j]
 * corresponds to the resulting state of transitioning from state i with character
 * START + j.
 * Remember that the i-th entry of the pi function does not correspond to the i-th state
 * of the automata.
 *
 * START - first character of the alphabet
 * ALPH - alphabet size
 *
 * Complexity: O(n * ALPH)
 */

const int ALPH = 26;
const char START = 'a';

struct kmpAutomata {
    vector<int> pi;
    vector<array<int, ALPH> > kmp;
    string s;
    int go(int i, int j){
        if(kmp[i][j] != -1) return kmp[i][j];
        int ans;
        if(s[i] == j + START) ans = i + 1;
        else if(i == 0) ans = 0;
        else ans = go(pi[i-1], j);
        return kmp[i][j] = ans;
    }
    kmpAutomata(string _s) : s(_s) {
        s += '#';
        int n = sz(s);
        pi.resize(n);
        pi[0] = 0;
        int k = 0;
        for(int i = 1; i < n; i++) {
            while(k && s[i] != s[k])
                k = pi[k-1];
            if(s[i] == s[k]) k++;
            pi[i] = k;
        }
        kmp.resize(n);
        for(int i = 0; i < n; i++)
            for(int j = 0; j < ALPH; j++)
                kmp[i][j] = -1;
        for(int i = 0; i < n; i++)
            for(int j = 0; j < ALPH; j++)
                go(i, j);
    }
};
```

suffix_array.cpp

```
/* Computes the Suffix Array of a set of strings.
 *
 * v - (input) vector of strings for which the suffix array will be built.
 * s - concatenation of the strings in v divided by the character '$'
 * n - suffix array size.
 * sa - suffix array. sa[i] means that the suffix s[sa[i], n) is the lexicografical i-th
 *      suffix of s
 * lcp - array of longest common prefixes. Must call computeLcp to calculate the entries
 *      of this array. lcp[i] means that the longest common prefix between s[sa[i-1], n)
 *      and s[sa[i], n) has length lcp[i]
 *
 * Complexity: O(n logn) time for building the suffix array.
 *             O(n) time for computing the lcp array.
 *             O(n) memory.
 */

struct suffixArray {
    string s;
    vector<int> sa, lcp;
    int n;
    void cSort(int k, vector<int>& ra) {
        int maxi = max(300, n);
        vector<int> c(maxi, 0), temp_sa(n);
        for(int i = 0; i < n; i++)
            c[i + k < n ? ra[i + k] : 0]++;
        for(int i = 1; i < maxi; i++)
            c[i] += c[i - 1];
        for(int i = n - 1; i >= 0; i--)
            temp_sa[--c[sa[i] + k < n ? ra[sa[i] + k] : 0]] = sa[i];
        sa.swap(temp_sa);
    }
    suffixArray() {}
    suffixArray(vector<string>& v) {
        for(auto str: v) {
            s += str;
            s += '$';
        }
        n = sz(s);
        sa.resize(n);
        vector<int> ra(n), temp_ra(n);
        for(int i = 0; i < n; i++) {
            ra[i] = s[i];
            sa[i] = i;
        }
        for(int k = 1; k < n; k <<= 1) {
            cSort(k, ra);
            cSort(0, ra);
            int r = temp_ra[sa[0]] = 0;
            for(int i = 1; i < n; i++)
```

```
                temp_ra[sa[i]] = (ra[sa[i]] == ra[sa[i - 1]]
                                    && ra[sa[i] + k] == ra[sa[i - 1] + k]) ? r : ++r;
            ra.swap(temp_ra);
            if(r == n - 1) break;
        }
    }
    void computeLcp() {
        vector<int> phi(n);
        int k = 0;
        phi[sa[0]] = -1;
        for(int i = 1; i < n; i++)
            phi[sa[i]] = sa[i-1];
        vector<int> plcp(n);
        for(int i = 0; i < n - 1; i++) {
            if(phi[i] == -1) {
                plcp[i] = 0;
                continue;
            }
            while(s[i + k] == s[phi[i] + k]) k++;
            plcp[i] = k;
            k = max(k - 1, 0);
        }
        lcp.resize(n);
        for(int i = 0; i < n; i++)
            lcp[i] = plcp[sa[i]];
    }
};
```

min_cyclic_string.cpp

```
/* Returns the start index of the minimum cyclic string
 * Uses Lyndon Factorization
 *
 * Complexity: O(n)
 */
int min_cyclic_string(string s) {
    s += s;
    int n = s.size();
    int i = 0, res = 0;

    while (i < n / 2) {
        res = i;
        int j = i + 1, k = i;

        while (j < n && s[k] <= s[j]) {
            if (s[k] < s[j]) {
                k = i;
            } else {
                k++;
            }
            j++;
        }

        while (i <= k) {
            i += j - k;
        }
    }

    return res;
}
```

Utils

random.cpp, k-mino.cpp

random.cpp

```
/* Random number generator
 *
 * Complexity: O(1)
 */
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
int x = rng() % n; // to generante random number

// to generate a uniform random number in the range [a, b]
ll random(ll a, ll b) {
    return uniform_int_distribution<ll> (a, b) (rng);
}
```

k-mino.cpp

```
/* Example code to generate k-minos
 *
 * Complexity: not efficient
 */
int dx[4] = {0, 0, -1, 1};
int dy[4] = {-1, 1, 0, 0};

struct Tab {
    int w, h;
    int tab[10][10];

    Tab() {
        w = 0;
        h = 0;
        for (int i = 0; i < 10; i++) {
            for (int j = 0; j < 10; j++) {
                tab[i][j] = 0;
            }
        }
    }

    Tab(const Tab &t) {
        w = t.w;
        h = t.h;
        for (int i = 0; i < 10; i++) {
            for (int j = 0; j < 10; j++) {
                tab[i][j] = t.tab[i][j];
            }
        }
    }

    void shiftDown() {
        for (int i = h; i > 0; i--) {
            for (int j = 0; j < w; j++) {
                tab[i][j] = tab[i-1][j];
            }
        }
        for (int j = 0; j < w; j++) tab[0][j] = 0;
        h++;
    }

    void shiftRight() {
        for (int j = w; j > 0; j--) {
            for (int i = 0; i < h; i++) {
                tab[i][j] = tab[i][j-1];
            }
        }
        for (int i = 0; i < h; i++) tab[i][0] = 0;
        w++;
    }

    pair<ll, ll> getHash() {
        ll h[2];
        for (int k = 0; k < 2; k++) {
            h[k] = 0;
            for (int i = 0; i < 5; i++) {
                for (int j = 0; j < 10; j++) {
                    h[k] *= 2;
                    h[k] += tab[5*k + i][j];
                }
            }
        }

        return {h[0], h[1]};
    }

    void print() {
        for (int i = 0; i < h; i++) {
            for (int j = 0; j < w; j++) {
                cout << (tab[i][j] ? 'O' : ' ');
            }
            cout << "\n";
        }
        cout << "\n";
    }
};

vector<Tab> ps[11];

void rec(int k) {
    if (k == 1) {
        Tab t;
        t.w = 1;
        t.h = 1;
        t.tab[0][0] = 1;
        ps[k].push_back(t);
        return;
    }

    set<pair<ll, ll>> hashes;
    rec(k-1);

    for (auto &t : ps[k-1]) {
        for (int i = -1; i <= t.h; i++) {
            for (int j = -1; j <= t.w; j++) {
                if (i >= 0 && i < 10 && j >= 0 && j < 10 && t.tab[i][j] == 1) continue;
                bool any = false;
                for (int a = 0; a < 4; a++) {
                    int ai = i + dx[a];
                    int aj = j + dy[a];
                    if (ai < 0 || ai >= 10 || aj < 0 || aj >= 10 || t.tab[ai][aj] == 0) continue;
                    any = true;
                    break;
                }
                if (any) {
                    Tab nt(t);
                    if (i == -1) {
                        i++;
                        nt.shiftDown();
                    } else if (i == t.h) {
                        nt.h++;
                    }
                    if (j == -1) {
                        j++;
                        nt.shiftRight();
                    } else if (j == t.w) {
                        nt.w++;
                    }
                    nt.tab[i][j] = 1;
                    pair<ll, ll> h = nt.getHash();
                    if (hashes.find(h) == hashes.end()) {
                        ps[k].push_back(nt);
                        hashes.insert(h);
                    }
                }
            }
        }
    }
}

int cx[51][51];

void solve() {
    int n, m;
    cin >> n >> m;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cin >> cx[i][j];
        }
    }
    int res = 0;
    for (auto &p : ps[m]) {
        vector<pair<int, int>> pos;
        for (int i = 0; i < p.h; i++) {
            for (int j = 0; j < p.w; j++) {
                if (p.tab[i][j]) pos.push_back({i, j});
            }
        }
        for (int i = 0; i < n - p.h + 1; i++) {
            for (int j = 0; j < n - p.w + 1; j++) {
                int cand = 0;
                for (int k = 0; k < m; k++) {
                    int x = i + pos[k].first;
                    int y = j + pos[k].second;
                    cand += cx[x][y];
                }
                res = max(res, cand);
            }
        }
    }
    cout << res << "\n";
}

int main() {
    FASTIO;
    rec(10);
    solve();
    return 0;
}
```