Contents

DataStructure/2d_pref_sum.cpp

```
#include <bits/stdc++.h>
using namespace std;
template<typename T>
struct pref_sum_2d {
    int n, m;
    vector<vector<T>> sum;
    template<typename U>
    pref_sum_2d(const vector<vector<U>>& a)
        : n((int)a.size()), m((int)a[0].size()), sum(n+1,
vector<T>(m+1)) {
            for (int i = 0; i < n; i++)
                for (int j = 0; j < m; j++) {
                    sum[i+1][j+1]=a[i][j] + sum[i][j+1] + sum[i+1][j] -
sum[i][j];
                }
        }
    T query(int x1, int y1, int x2, int y2) {
        return sum[x2+1][y2+1] - sum[x2+1][y1] - sum[x1][y2+1] + sum[x1]
[y1];
    }
};
```

DataStructure/fenwick.cpp

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

using ll = long long;
template <typename T> struct fenwick {
    int n; vector<T> t;
    fenwick(int n) : n(n), t(n + 1) {}
    void add(int i, T x) {
        assert(i >= 0 && i < n);
        for (i++; i <= n; i += i & -i) {
            t[i] += x;
        }
    }
    // change return type if needed
    T query(int i) {
        assert(i >= -1 && i < n);
        T res{};
        for (i++; i > 0; i -= i & -i)
            res += t[i];
        return res;
    }
    // change return type if needed
    T query(int l, int r) {
        assert(l >= 0 && l <= r && r < n);
        return query(r) - query(l - 1);
    }
    int search(T prefix) { // finds first pos s.t. sum(0, pos)>=prefix
        int pos = 0;
        T sum = 0;
        for (int i = __lg(n); i >= 0; i--) {
            // could change < to <= to make it find upper bound
            if (int nxt = pos + (1 << i); nxt <= n && (sum + t[nxt] <
prefix)) {
                pos = nxt;
                sum += t[pos];
            }
        }
        return pos;
    }
};

// fenwick tree with range update and range sum query
struct fenwick_rg {
    int n;
    vector<ll> sum1, sum2;
    fenwick_rg(int n) : n(n), sum1(n + 1), sum2(n + 1) {}
    void add(int i, int x) {
        assert(i >= 0 && i < n);
        i++;
        ll v = (ll)i * x;
        for (; i <= n; i += i & -i)
            sum1[i] += x, sum2[i] += v;
    }
    void add(int l, int r, int x) {
        assert(l >= 0 && l <= r && r < n);
        add(l, x);
        if (r + 1 < n) add(r + 1, -x);
    }
    ll query(int p) {
        assert(p >= -1 && p < n);
        p++;
        ll res{};
        for (int i = p; i; i -= i & -i)
            res += (p + 1) * sum1[i] - sum2[i];
        return res;
    }
    ll query(int l, int r) {
        assert(l >= 0 && l <= r && r < n);
        return query(r) - query(l - 1);
    }
};
```

DataStructure/lazy_segtree.cpp

```
// segment tree with lazy propagation
#include<bits/stdc++.h>
using namespace std;

struct lazyseg {
    using s = int;
    using f = int;

    s e() {}
    s op(const s& x, const s& y) {}
    f id() {}
    f comp(const f& neo, const f& old) {}
    s mp(f, s) {}

    int n;
    vector<s> d;
    vector<f> lz;
    explicit lazyseg(int n) : lazyseg(vector<s>(n, e())) {}
    lazyseg(const vector<s> &v) : n((int)size(v)), d(4 * n), lz(4 * n,
id()) {
        build(1, 0, n - 1, v);
    }
    void pull(int k) { d[k] = op(d[k * 2], d[k * 2 + 1]); }
    void build(int k, int l, int r, const vector<s>& v) {
        if (l == r) {
            d[k] = v[l];
            return;
        }
        int mid = (l + r) / 2;
        build(k * 2, l, mid, v);
        build(k * 2 + 1, mid + 1, r, v);
        pull(k);
    }
    void all_apply(int k, f f) {
        d[k] = mp(f, d[k]);
        lz[k] = comp(f, lz[k]);
    }
    void push(int k) {
        all_apply(k * 2, lz[k]);
        all_apply(k * 2 + 1, lz[k]);
        lz[k] = id();
    }
    void apply(int k, int ql, int qr, int l, int r, f x) {
        if (r < ql || l > qr) return;
        if (ql <= l && qr >= r) {
            return all_apply(k, x);
        }
        push(k);
        int mid = (l + r) / 2;
        apply(k * 2, ql, qr, l, mid, x);
        apply(k * 2 + 1, ql, qr, mid + 1, r, x);
        pull(k);
    }
    s get(int k, int ql, int qr, int l, int r) {
        if (qr < l || ql > r) return e();
        if (ql <= l && qr >= r) return d[k];
        push(k);
        int mid = (l + r) / 2;
        return op(get(k * 2, ql, qr, l, mid), get(k * 2 + 1, ql, qr, mid
+ 1, r));
    }
    void apply(int l, int r, f x) {
        if (r < l) return;
        assert(l >= 0 && l <= r && r < n);
        apply(1, l, r, 0, n - 1, x);
    }
    s get(int i) {
        assert(i >= 0 && i < n);
        return get(1, i, i, 0, n - 1);
    }
    s get(int l, int r) {
        assert(l >= 0 && l <= r && r < n);
        return get(1, l, r, 0, n - 1);
    }
    template<class G> int max_right(int ql, G f) {
        assert(0 <= ql && ql <= n);
        assert(f(e()));
        s sum = e();
        auto rec = [&](auto& slf, int k, int l, int r) {
            if (s s = op(sum, d[k]); l >= ql && f(s)) {
                sum = s;
                return r;
            }
            if (l == r) return l - 1;
            push(k);
            int mid = (l + r) / 2;
            if (ql <= mid) {
                int res = slf(slf, k * 2, l, mid);
                if (res != mid) return res;
            }
            return slf(slf, k * 2 + 1, mid + 1, r);
        };
        return rec(rec, 1, 0, n - 1);
    }
    template<class G> int min_left(int qr, G f) {
        assert(-1 <= qr && qr < n);
        assert(f(e()));
        s sum = e();
        auto rec = [&](auto& slf, int k, int l, int r) {
            if (s s = op(d[k], sum); r <= qr && f(s)) {
                sum = s;
                return l;
            }
            if (l == r) return l + 1;
            push(k);
            int mid = (l + r) / 2;
            if (qr > mid) {
                int res = slf(slf, k * 2 + 1, mid + 1, r);
                if (res != mid + 1) return res;
            }
            return slf(slf, k * 2, l, mid);
        };
        return rec(rec, 1, 0, n - 1);
    }
};
```

DataStructure/line_container.cpp

```
#include <bits/stdc++.h>
using namespace std;
/**
 * Credit: https://github.com/kth-competitive-programming/kactl/blob/
main/content/data-structures/LineContainer.h
 * Author: Simon Lindholm
 * Date: 2017-04-20
 * License: CC0
 * Source: own work
 * Description: Container where you can add lines of the form kx+m, and
query
 * maximum values at points x. Useful for dynamic programming (``convex
hull
 * trick''). Time: O(\log N) Status: stress-tested
 */


using ll = long long;

struct Line {
  mutable ll k, m, p;
  bool operator<(const Line &o) const { return k < o.k; }
  bool operator<(ll x) const { return p < x; }
};

struct LineContainer : multiset<Line, less<>> {
  // (for doubles, use inf = 1/.0, div(a,b) = a/b)
  static const ll inf = LLONG_MAX;
  ll div(ll a, ll b) { // floored division
    return a / b - ((a ^ b) < 0 && a % b);
  }
  bool isect(iterator x, iterator y) {
    if (y == end()) return x->p = inf, 0;
    if (x->k == y->k) x->p = x->m > y->m ? inf : -inf;
    else x->p = div(y->m - x->m, x->k - y->k);
    return x->p >= y->p;
  }
  void add(ll k, ll m) {
    auto z = insert({k, m, 0}), y = z++, x = y;
    while (isect(y, z)) z = erase(z);
    if (x != begin() && isect(--x, y))
      isect(x, y = erase(y));
    while ((y = x) != begin() && (--x)->p >= y->p)
      isect(x, erase(y));
  }
  ll query(ll x) {
    assert(!empty());
    auto l = *lower_bound(x);
    return l.k * x + l.m;
  }
};
```

DataStructure/monotonic_dp_hull.cpp

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

// monotonic_dp_hull enables you to do the following two operations in
amortized O(1) time:
// 1. Insert a line (k, b) into the structure. k must be non-decreasing.
// 2. For any value of x, query the maximum value of k * x + b. x must
be non-decreasing.
// Note:
// 1. if slope and/or query is non-increasing, change position of
operation
// 2. if slope and/or query is in arbitrary order, use line_container
instead which has complexity of O(log n) per operation
struct monotonic_dp_hull {
    struct line {
        ll k, b;
        ll eval(ll x) { return k * x + b; }
    };

    bool bad(const line &a, const line &b, const line &c) {
        return (c.b - a.b) * (a.k - b.k) <= (b.b - a.b) * (a.k - c.k);
    }

    deque<line> lines;

    void insert(ll k, ll b) {
        assert(lines.empty() || k > lines.back().k); // ensure slope is
monotonic
        line cur{k, b};
        while (lines.size() >= 2 && bad(*(lines.rbegin() + 1),
lines.back(), cur))
            lines.pop_back();
        lines.push_back(cur);
    }

    ll query(ll x) {
        assert(!lines.empty());
        while (lines.size() >= 2 && lines[0].eval(x) <=
lines[1].eval(x))
            lines.pop_front();
        return lines[0].eval(x);
    }
};
```

DataStructure/persistent_seg.cpp

```
//find the nth biggest number
#include<bits/stdc++.h>
struct PST {
    int n, tot=0;
    vector<int> lc, rc, sum, roots; // left child, right child
    PST(int n_) : n(n_), lc(n<<5), rc(n<<5), sum(n<<5), roots(1) { //
change the size to n<<6 if there are 2*n modification
        build(0, n-1, roots[0]); // the initial root node is 1!
    }
    void pushup(int rt) {
        sum[rt] = sum[lc[rt]] + sum[rc[rt]];
    }
    void build(int l, int r, int& rt) {
        rt = ++tot;
        if (l == r) return;
        int mid = (l + r) >> 1;
        build(l, mid, lc[rt]);
        build(mid + 1, r, rc[rt]);
        pushup(rt);
    }
    void update(int pos, int val, int l, int r, int old, int& rt) {
        rt = ++tot;
        lc[rt] = lc[old];
        rc[rt] = rc[old];
        if (l == r) {
            sum[rt] = sum[old] + val;
            return;
        }
        int mid = (l + r) >> 1;
        if (pos <= mid) update(pos, val, l, mid, lc[old], lc[rt]);
        else update(pos, val, mid + 1, r, rc[old], rc[rt]);
        pushup(rt);
    }
    int update(int pos, int val) { // return the root of the new version
        int new_root;
        update(pos, val, 0, n-1, roots.back(), new_root);
        roots.push_back(new_root);
        return new_root;
    }
    int query(int u, int v, int l, int r, int k) {
        if (l==r) return l;
        int mid=(l+r)/2, x=sum[lc[v]]-sum[lc[u]];
        if (k<=x) return query(lc[u], lc[v], l, mid, k);
        return query(rc[u], rc[v], mid+1, r, k-x);
    }
};
int main(){
    int n, q;
    cin>>n>>q;
    vector<int> a(n);
    for (auto& x : a) cin>>x;
    auto comp=a;
    sort(comp.begin(), comp.end());
    comp.erase(unique(comp.begin(), comp.end()), comp.end());
    PST tr(comp.size());
    vector<int> roots(n+1);
    roots[0]=1;
    for (int i=0; i<n; i++) {
        int p=lower_bound(comp.begin(), comp.end(), a[i])-comp.begin();
        roots[i+1]=tr.update(p, 1);
    }
    while (q--) {
        int l, r, k;
        cin>>l>>r>>k;
        cout<<comp[tr.query(roots[l-1], roots[r], 0, comp.size()-1,
k)]<<'\n';
    }
}
```

DataStructure/segtree.cpp

```
#include <vector>
using namespace std;

template<class s, auto op, auto e>
struct segtree {
    int n;
    vector<s> d;
    segtree(int n) : n(n), d(4 * n) {
        build(1, 0, n - 1, vector<s>(n, e()));
    }
    segtree(const vector<s> &v) : n((int)v.size()), d(4 * n) {
        build(1, 0, n - 1, v);
    }
    void pull(int k) { d[k] = op(d[k * 2], d[k * 2 + 1]); }
    void build(int p, int l, int r, const vector<s> &v) {
        if (l == r) {
            d[p] = v[l];
            return;
        }
        int mid = (l + r) >> 1;
        build(p * 2, l, mid, v);
        build(p * 2 + 1, mid + 1, r, v);
        pull(p);
    }
    void set(int p, int i, s x, int l, int r) {
        if (l == r) {
            d[p] = x;
            return;
        }
        int m = (l + r) / 2;
        if (i <= m) set(p * 2, i, x, l, m);
        else set(p * 2 + 1, i, x, m + 1, r);
        pull(p);
    }
    s get(int p, int ql, int qr, int l, int r) {
        if (ql > r || qr < l) return e();
        if (ql <= l && qr >= r) return d[p];
        int m = (l + r) / 2;
        return op(get(p * 2, ql, qr, l, m), get(p * 2 + 1, ql, qr, m +
1, r));
    }
    s get(int i) {
        assert(i >= 0 && i < n);
        return get(1, i, i, 0, n - 1);
    }
    s get(int l, int r) {
        assert(l >= 0 && l <= r && r < n);
        return get(1, l, r, 0, n - 1);
    }
    void set(int i, s x) {
        assert(i >= 0 && i < n);
        set(1, i, x, 0, n - 1);
    }
    // return the largest r such that f(op(d[ql], ..., d[r])) is true
    template<class G> int max_right(int ql, G f) {
        assert(0 <= ql && ql <= n);
        assert(f(e()));
        s sum = e();
        auto rec = [&](auto& slf, int k, int l, int r) {
            if (l >= ql) {
                auto ss = op(sum, d[k]);
                if (f(ss)) {
                    sum = op(sum, d[k]);
                    return r;
                }
            }
            if (l == r) return l - 1;
            int mid = (l + r) / 2;
            if (ql <= mid) {
                int res = slf(slf, k * 2, l, mid);
                if (res != mid) return res;
            }
            return slf(slf, k * 2 + 1, mid + 1, r);
        };
        return rec(rec, 1, 0, n - 1);
    }

    // return the smallest l such that f(op(d[l], ..., d[qr])) is true
    template<class G> int min_left(int qr, G f) {
        assert(-1 <= qr && qr < n);
        assert(f(e()));
        if (qr == -1) return 0;
        s sum = e();
        auto rec = [&](auto& slf, int k, int l, int r) {
            if (r <= qr) {
                s ss = op(d[k], sum);
                if (f(ss)) {
                    sum = op(d[k], sum);
                    return l;
                }
            }
            if (l == r) return l + 1;
            int mid = (l + r) / 2;
            if (qr > mid) {
                int res = slf(slf, k * 2 + 1, mid + 1, r);
                if (res != mid + 1) return res;
            }
            return slf(slf, k * 2, l, mid);
        };
        return rec(rec, 1, 0, n - 1);
    }
};
```

DataStructure/sliding_window.cpp

```
template<typename T, typename compare = less<T>>
struct sliding_window {
    int k; // width of the window
    deque<pair<int, T>> q;
    compare cmp;
    sliding_window(int k_) : k(k_), cmp() {}
    void add(int i, T x) {
        while (!q.empty() && !cmp(q.back().second, x)) q.pop_back();
        q.emplace_back(i, x);
        while (q.front().first <= i - k) q.pop_front();
    }
    T get() { return q.front().second; }
};
```

DataStructure/sparse-table.cpp

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

int main() {
    int n;
    vector<int> a(n);

    int logn = __lg(n);
    vector v(logn + 1, vector<int>(n));
    v[0] = a;
    for (size_t i = 1; i <= logn; i++)
        for (size_t j = 0; j + (1 << i) - 1 < n; j++)
            v[i][j] = max(v[i - 1][j], v[i - 1][j + (1 << (i - 1))]);

    // [l, r]
    auto query = [&](int l, int r) {
        assert(l <= r);
        int s = __lg(r - l + 1);
        return max(v[s][l], v[s][r - (1 << s) + 1]);
    };
}

namespace st { // 2d sparse table
    using T = int;
    int n, m, logn, logm;
    static const int N = 1e3 + 5;
    T t[13][13][N][N]; // array layout matches loop order to ensure
efficiency

    template<typename U>
    void init(const vector<vector<U>>& val) {
        n = ((int)val.size()), m = ((int)val[0].size()),
        logn = (__lg(n)), logm = (__lg(m));
        for (int i = 0; i < n; i++) for (int j = 0; j < m; j++) t[0][0]
[i][j] = val[i][j];
```

```
        for (int i = 0; i <= logn; i++)
            for (int j = 0; j <= logm; j++) {
                if (i == 0 && j == 0) continue;
                for (int row = 0; row + (1 << i) - 1 < n; row++) {
                    for (int col = 0; col + (1 << j) - 1 < m; col++) {
                        // auto &v = t[row][col];
                        if (i == 0)
                            t[i][j][row][col] = min(t[i][j - 1][row]
[col], t[i][j - 1][row][col + (1 << (j - 1))]);
                        if (j == 0)
                            t[i][j][row][col] = min(t[i - 1][j][row]
[col], t[i - 1][j][row + (1 << (i - 1))][col]);
                        else
                            t[i][j][row][col] = min(t[i][j - 1][row]
[col], t[i][j - 1][row][col + (1 << (j - 1))]);
                    }
                }
            }
    }
    T query(int x1, int x2, int y1, int y2) {
        assert(n!=0 && m!=0);
        assert(x1 <= x2);
        assert(y1 <= y2);
        assert(x1 >= 0 && x1 < n);
        assert(x2 >= 0 && x2 < n);
        assert(y1 >= 0 && y1 < m);
        assert(y2 >= 0 && y2 < m);
        int kx = __lg(x2 - x1 + 1), ky = __lg(y2 - y1 + 1);
        return min(
            {t[kx][ky][x1][y1], t[kx][ky][x2 - (1 << kx) + 1][y1],
             t[kx][ky][x1][y2 - (1 << ky) + 1],
             t[kx][ky][x2 - (1 << kx) + 1][y2 - (1 << ky) + 1]});
    }

};
```

DataStructure/treap_split.cpp

```
// using treap to maintain a sequence that support multiple operation,
index
// 0-based index, change pull(), add(), pushdown() according to the
problem
mt19937
gen(chrono::high_resolution_clock::now().time_since_epoch().count());
template <typename T> struct Treap {
    struct node {
        int ch[2], sz;
        unsigned k;
        T d, sum, lazy;
        node(T d_, int z = 1)
            : sz(z), k((unsigned)gen()), d(d_), sum(d), lazy() {
            ch[0] = ch[1] = 0;
        }
    };
    vector<node> nodes;
    int root=0;
    Treap(int size = 2e5) {
        nodes.reserve(size);
        nodes.emplace_back(0, 0);
    }
    inline int &ch(int rt, int r) { return nodes[rt].ch[r]; }
    int new_node(const T &d) {
        int id = (int)nodes.size();
        nodes.push_back(node(d));
        return id;
    }
    int pull(int rt) {
        node &n = nodes[rt];
        n.sz = 1 + nodes[n.ch[0]].sz + nodes[n.ch[1]].sz;
        n.sum = n.d + nodes[n.ch[0]].sum + nodes[n.ch[1]].sum;
        return rt;
    }
    void add(int rt, const T &d) {
        node &n = nodes[rt];
        n.lazy = n.lazy + d;
        n.d = n.d + d;
        n.sum = n.sum + d * n.sz;
    }
    void pushdown(int rt) {
        node &n = nodes[rt];
        if (n.lazy) {
            add(n.ch[0], n.lazy);
            add(n.ch[1], n.lazy);
            n.lazy = T();
        }
    }
    int merge(int tl, int tr) {
        if (!tl) return tr;
        if (!tr) return tl;
        if (nodes[tl].k < nodes[tr].k) {
            pushdown(tl);
            ch(tl, 1) = merge(ch(tl, 1), tr);
            return pull(tl);
        } else {
            pushdown(tr);
            ch(tr, 0) = merge(tl, ch(tr, 0));
            return pull(tr);
        }
    }
    void split_by_size(int rt, int k, int &x, int &y) { // split out
first k element
        if (!rt) { x = y = 0; return; }
        pushdown(rt);
        if (k <= nodes[ch(rt, 0)].sz) {
            y = rt;
            split_by_size(ch(rt, 0), k, x, ch(rt, 0));
        } else {
            x = rt;
            split_by_size(ch(rt, 1), k - nodes[ch(rt, 0)].sz - 1, ch(rt,
1), y);
        }
        pull(rt);
    }
    void split_by_val(int rt, const T& target, int& x, int& y) {// split
into two sets such that one contains <=k and other contains >k
        if (!rt) { x=y=0; return; }
        pushdown(rt);
        if (target < nodes[rt].d) {
            y = rt;
            split_by_val(ch(rt, 0), target, x, ch(rt, 0));
        } else {
            x = rt;
            split_by_val(ch(rt, 1), target, ch(rt, 1), y);
        }
        pull(rt);
    }
    void remove(int &rt) { rt = 0; }
    // interface
    int size() { return nodes[root].sz; }
    const T& operator[](int k) {
        assert(k>=0 && k<size());
        int x, y, z;
        split_by_size(root, k+1, y, z);
        split_by_size(y, k, x, y);
        root = merge(merge(x, y), z);
        return nodes[y];
    }
    void insert(int k, T v) { // insert at kth position
        assert(k>=0 && k<=size());
        int l, r;
        split_by_size(root, k, l, r);
        int rt = new_node(v);
        root = merge(merge(l, rt), r);
    }
    void erase(int l, int r) {
        assert(l>=0 && l<=r && r<size());
        int x, y, z;
        split_by_size(root, r + 1, y, z);
        split_by_size(y, l, x, y);
        remove(y);
        root = merge(x, z);
    }
    void range_add(int l, int r, T v) {
        assert(l>=0 && l<=r && r<size());
        int x, y, z;
        split_by_size(root, r + 1, y, z);
        split_by_size(y, l, x, y);
        add(y, v);
        root = merge(merge(x, y), z);
    }
    T getsum(int l, int r) {
        assert(l>=0 && l<=r && r<size());
        int x, y, z;
        split_by_size(root, r + 1, y, z);
        split_by_size(y, l, x, y);
        T ret = nodes[y].sum;
        root = merge(merge(x, y), z);
        return ret;
    }
};
```

DataStructure/union_find.cpp

```
struct UF {
    int n;
    vector<int> pa; // parent or size, positive number means parent,
negative number means size
    explicit UF(int _n) : n(_n), pa(n, -1) {}
    int find(int x) {
        assert(0 <= x && x < n);
        return pa[x] < 0 ? x : pa[x]=find(pa[x]);
    }
    bool join(int x, int y) {
        assert(0 <= x && x < n && 0 <= y && y < n);
        x=find(x), y=find(y);
        if (x==y) return false;
        if (-pa[x] < -pa[y]) swap(x, y); // size of x is smaller than
size of y
        pa[x]+=pa[y];
        pa[y]=x;
        return true;
    }
    int size(int x) {
        assert(0 <= x && x < n);
        return -pa[x];
    }
    vector<vector<int>> groups() {
        vector<int> leader(n);
        for (int i=0; i<n; i++) leader[i]=find(i);
        vector<vector<int>> res(n);
        for (int i=0; i<n; i++) {
            res[leader[i]].push_back(i);
        }
        res.erase(remove_if(res.begin(), res.end(),
                    [](const vector<int>& v) { return v.empty(); }),
res.end());
        return res;
    }
};
```

DataStructure/wavelet-tree.hpp

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

using namespace std;

struct WaveletTree {
    using iter = vector<int>::iterator;
    vector<vector<int>> c;
    const int SIGMA;

    WaveletTree(vector<int> a, int sigma): c(sigma*2), SIGMA(sigma) {
        build(a.begin(), a.end(), 0, SIGMA, 1);
    }

    void build(iter begin, iter end, int l, int r, int u) {
        if(r - l == 1) return;
        int m = (l + r) / 2;

        c[u].reserve(end - begin + 1);
        c[u].push_back(0);
        auto f = [=](int i) { return i < m; };
        for (auto it = begin; it != end; ++it) {
            c[u].push_back(c[u].back() + f(*it));
        }

        auto p = stable_partition(begin, end, f);
        build(begin, p, l, m, 2 * u);
        build(p, end, m, r, 2 * u + 1);
    }

    // occurrences of val in position[0, i)
    int rank(int val, int i) const {
        if(val < 0 or val >= SIGMA) return 0;

        int l = 0, r = SIGMA, u = 1;
        while(r - l > 1) {
            int m = (l + r) / 2;
            if(val < m) {
                i = c[u][i], r = m;
                u = u * 2;
            } else {
                i -= c[u][i], l = m;
                u = u * 2 + 1;
            }
        }
        return i;
    }

    int quantile(int k, int i, int j) const {
        assert(k > 0 && k <= j - i);
        int l = 0, r = SIGMA, u = 1;
        while(r - l > 1) {
            int m = (l + r) / 2;
            int ni = c[u][i], nj = c[u][j];
            if(k <= nj - ni) {
                i = ni, j = nj, r = m;
                u = 2 * u;
            } else {
                k -= nj - ni;
                i -= ni, j -= nj, l = m;
                u = 2 * u + 1;
            }
        }
        return l;
    }
};
```

DataStructure/xor_basis.hpp

```
#include <bits/stdc++.h>
using namespace std;
template <typename T> struct XorBasis {
    static constexpr int B = 8 * sizeof(T);
    T basis[B]{};
    int sz = 0;

    void insert(T x) {
        for (int i = B - 1; i >= 0; i--) {
            if (x >> i == 0) continue;
            if (!basis[i]) {
                basis[i] = x;
                sz++;
                break;
            }
            x ^= basis[i];
        }
    }

    bool is_in(T x) {
        for (int i = B - 1; i >= 0; i--) {
            if (x >> i == 0) continue;
            if (!basis[i]) return false;
            x ^= basis[i];
        }
        return true;
    }

    T max_value(T start = 0) {
        for (int i = B - 1; i >= 0; i--) {
            if (basis[i]) {
                start = max(start, start ^ basis[i]);
            }
        }
        return start;
    }

    // return the kth (0-indexed) smallest element in the vector space
    T kth(long long k) {
        assert(k >= 0 && k < (1LL << sz));
        T ans{};
        int b = sz - 1;
        for (int i = B - 1; i >= 0; i--) {
            if (basis[i]) {
                if (k >> b & 1) {
                    ans = max(ans, ans ^ basis[i]);
                } else {
                    ans = min(ans, ans ^ basis[i]);
                }
                b--;
            }
        }
        return ans;
    }
};
```

Geometry/angle.h

```
double DEG_to_RAD(double d) { return d*M_PI/180.0; }
double RAD_to_DEG(double r) { return r*180.0/M_PI; }
double rad(P p1,P p2){
  return atan2l(p1.det(p2),p1.dot(p2));
}
bool inAngle(P a, P b, P c, P p) {
  assert(crossOp(a,b,c) != 0);
  if (crossOp(a,b,c) < 0) swap(b,c);
  return crossOp(a,b,p) >= 0 && crossOp(a,c,p) <= 0;
}
double angle(P v, P w) {
  return acos(clamp(v.dot(w) / v.abs() / w.abs(), -1.0, 1.0));
}
double orientedAngle(P a, P b, P c) { // BAC
  if (crossOp(a,b,c) >= 0) return angle(b-a, c-a);
  else return 2*M_PI - angle(b-a, c-a);
}
```

Geometry/circle.h

```
// double chord(double r, double ang) return sqrt(2*r*r*(1-
cos(ang))); // or 2*r*sin(ang/2)
// double secarea(double r, double ang) {return (ang/2)*(r*r);} // rad
// double segarea(double r, double ang) {return secarea(r, ang) -
r*r*sin(ang)/2;}
int type(P o1,double r1,P o2,double r2){
  double d = o1.distTo(o2);
  if(cmp(d,r1+r2) == 1) return 4; // outside each other
  if(cmp(d,r1+r2) == 0) return 3; // touch outside
  if(cmp(d,abs(r1-r2)) == 1) return 2; // one inside another
  if(cmp(d,abs(r1-r2)) == 0) return 1; // touch inside
  return 0;
}
vector<P> isCL(P o,double r,P p1,P p2){
  if (cmp(abs((o-p1).det(p2-p1)/p1.distTo(p2)),r)>0) return {};
  double x = (p1-o).dot(p2-p1), y = (p2-p1).abs2(), d = x * x - y *
((p1-o).abs2() - r*r);
  d = max(d,0.0); P m = p1 - (p2-p1)*(x/y), dr = (p2-p1)*(sqrt(d)/y);
  return {m-dr,m+dr}; //along dir: p1->p2
}
vector<P> isCC(P o1, double r1, P o2, double r2) { //need to check
whether two circles are the same
  double d = o1.distTo(o2);
  if (cmp(d, r1 + r2) == 1) return {};
  if (cmp(d,abs(r1-r2))==-1) return {};
  d = min(d, r1 + r2);
  double y = (r1 * r1 + d * d - r2 * r2) / (2 * d), x = sqrt(r1 * r1 - y
* y);
  P dr = (o2 - o1).unit();
  P q1 = o1 + dr * y, q2 = dr.rot90() * x;
  return {q1-q2,q1+q2};//along circle 1
}
vector<P> tanCP(P o, double r, P p) {
  double x = (p - o).abs2(), d = x - r * r;
  if (sign(d) <= 0) return {}; // on circle => no tangent
  P q1 = o + (p - o) * (r * r / x);
  P q2 = (p - o).rot90() * (r * sqrt(d) / x);
  return {q1-q2,q1+q2}; //counter clock-wise
}
vector<L> extanCC(P o1, double r1, P o2, double r2) {
  vector<L> ret;
  if (cmp(r1, r2) == 0) {
    P dr = (o2 - o1).unit().rot90() * r1;
    ret.push_back(L(o1 + dr, o2 + dr)), ret.push_back(L(o1 - dr, o2 -
dr));
  } else {
    P p = (o2 * r1 - o1 * r2) / (r1 - r2);
    vector<P> ps = tanCP(o1, r1, p), qs = tanCP(o2, r2, p);
    for(int i = 0; i < min(ps.size(),qs.size());i++)
ret.push_back(L(ps[i], qs[i])); //c1 counter-clock wise
  }
  return ret;
}
vector<L> intanCC(P o1, double r1, P o2, double r2) {
  vector<L> ret;
  P p = (o1 * r2 + o2 * r1) / (r1 + r2);
  vector<P> ps = tanCP(o1,r1,p), qs = tanCP(o2,r2,p);
  for(int i = 0; i < min(ps.size(),qs.size()); i++)
ret.push_back(L(ps[i], qs[i])); //c1 counter-clock wise
  return ret;
}
double areaCT(double r, P p1, P p2){
  vector<P> is = isCL(P(0,0),r,p1,p2);
  if(is.empty()) return r*r*rad(p1,p2)/2;
  bool b1 = cmp(p1.abs2(),r*r) == 1, b2 = cmp(p2.abs2(), r*r) == 1;
  if(b1 && b2){
    if(sign((p1-is[0]).dot(p2-is[0])) <= 0 &&
      sign((p1-is[0]).dot(p2-is[0])) <= 0)
    return r*r*(rad(p1,is[0]) + rad(is[1],p2))/2 + is[0].det(is[1])/2;
    else return r*r*rad(p1,p2)/2;
  }
  if(b1) return (r*r*rad(p1,is[0]) + is[0].det(p2))/2;
  if(b2) return (p1.det(is[1]) + r*r*rad(is[1],p2))/2;
  return p1.det(p2)/2;
}
P inCenter(P A, P B, P C) {
  double a = (B - C).abs(), b = (C - A).abs(), c = (A - B).abs();
  return (A * a + B * b + C * c) / (a + b + c);
}
P circumCenter(P a, P b, P c) {
  P bb = b - a, cc = c - a;
  double db = bb.abs2(), dc = cc.abs2(), d = 2 * bb.det(cc);
  return a - P(bb.y * dc - cc.y * db, cc.x * db - bb.x * dc) / d;
}
P othroCenter(P a, P b, P c) {
  P ba = b - a, ca = c - a, bc = b - c;
  double Y = ba.y * ca.y * bc.y,
  A = ca.x * ba.y - ba.x * ca.y,
  x0 = (Y + ca.x * ba.y * b.x - ba.x * ca.y * c.x) / A,
  y0 = -ba.x * (x0 - c.x) / ba.y + ca.y;
  return {x0, y0};
}
```

Geometry/geometry.h

```
typedef double T;
const double EPS = 1e-9;
inline int sign(double a) { return a < -EPS ? -1 : a > EPS; }
inline int cmp(double a, double b){ return sign(a-b); }
struct P {
  T x,y;
  P() {}
  P(T _x, T _y) : x(_x), y(_y) {}
  P operator+(P p) {return {x+p.x, y+p.y};}
  P operator-(P p) {return {x-p.x, y-p.y};}
  P operator*(T d) {return {x*d, y*d};}
  P operator/(T d) {return {x/d, y/d};} // only for floatingpoint
  bool operator<(P p) const {
    int c = cmp(x, p.x);
    if (c) return c == -1;
    return cmp(y, p.y) == -1;
  }
  bool operator==(P o) const{
    return cmp(x,o.x) == 0 && cmp(y,o.y) == 0;
  }
  double dot(P p) { return x * p.x + y * p.y; }
  double det(P p) { return x * p.y - y * p.x; }
  double distTo(P p) { return (*this-p).abs(); }
  double alpha() { return atan2(y, x); }
  void read() { cin>>x>>y; }
  void write() {cout<<"("<<x<<","<<y<<")"<<endl;}
  double abs() { return sqrt(abs2());}
  double abs2() { return x * x + y * y; }
  P rot90() { return P(-y,x);}
  P unit() { return *this/abs(); }
  int quad() const { return sign(y) == 1 || (sign(y) == 0 && sign(x) >=
0); }
  P rot(double an){ return {x*cos(an)-y*sin(an),x*sin(an) +
y*cos(an)}; }
};
#define cross(p1,p2,p3) ((p2.x-p1.x)*(p3.y-p1.y)-(p3.x-p1.x)*(p2.y-p1.
y))
#define crossOp(p1,p2,p3) sign(cross(p1,p2,p3))
bool isConvex(vector<P> p) {
  bool hasPos=false, hasNeg=false;
  for (int i=0, n=p.size(); i<n; i++) {
    int o = cross(p[i], p[(i+1)%n], p[(i+2)%n]);
    if (o > 0) hasPos = true;
    if (o < 0) hasNeg = true;
  }
  return !(hasPos && hasNeg);
}
bool half(P p) {
  assert(p.x != 0 || p.y != 0); // (0, 0) is not covered
  return p.y > 0 || (p.y == 0 && p.x < 0);
}
void polarSortAround(P o, vector<P> &v) {
  sort(v.begin(), v.end(), [&o](P v, P w) {
      return make_tuple(half(v-o), 0) <
        make_tuple(half(w-o), cross(o, v, w));
  });
}
P proj(P p1, P p2, P q) {
  P dir = p2 - p1;
  return p1 + dir * (dir.dot(q - p1) / dir.abs2());
}
P reflect(P p1, P p2, P q){
  return proj(p1,p2,q) * 2 - q;
}
// tested with https://open.kattis.com/problems/closestpair2
pair<P, P> closest(vector<P> v) {
  assert(sz(v) > 1);
  set <P> S;
  sort(v.begin(), v.end(), [](P a, P b) { return a.y < b.y; });
  pair<T, pair<P, P>> ret{(T)1e18, {P(), P()}};
  int j = 0;
  for(P p : v) {
    P d { 1 + (T) sqrt(ret.first), 0 };
    while(p.y - v[j].y >= d.x) S.erase(v[j++]);
    auto lo = S.lower_bound(p - d), hi = S.upper_bound(p + d);
    for(; lo != hi; ++lo) {
      ret = min(ret, {(p - (*lo)).abs2(), {*lo, p}});
    }
    S.insert(p);
  }
  return ret.second;
}
struct L {
  P ps[2]; P v; T c;
  L() {}
  P& operator[](int i) { return ps[i]; }
  // From direction vector v and offset c
  L(P v, T c) : v(v), c(c) {}
  // From equation ax+by=c
  L(T a, T b, T c) : v({b,-a}), c(c) {}
  // From points P and Q
  L(P p, P q) : v(q-p), c(cross(P(0, 0), v,p)) {
    ps[0] = p;
    ps[1] = q;
  }
  P dir() { return ps[1] - ps[0]; }
  bool include(P p) { return sign((ps[1] - ps[0]).det(p - ps[0])) > 0; }
  T side(P p) {return cross(P(0, 0), v,p)-c;}
  T dist(P p) {return abs(side(p)) / v.abs();}
  T sqDist(P p) {return side(p)*side(p) / (double)v.abs();}
  L perpThrough(P p) {return L(p, p + v.rot90());}
  bool cmpProj(P p, P q) {
    return v.dot(p) < v.dot(q);
  }
  L translate(P t) {return L(v, c + cross(P(0,0), v,t));}
  L shiftLeft(double dist) {return L(v, c + dist*v.abs());}
  L shiftRight(double dist) {return L(v, c - dist*v.abs());}
};
bool chkLL(P p1, P p2, P q1, P q2) {
  double a1 = cross(q1, q2, p1), a2 = -cross(q1, q2, p2);
  return sign(a1+a2) != 0;
}
P isLL(P p1, P p2, P q1, P q2) {
  double a1 = cross(q1, q2, p1), a2 = -cross(q1, q2, p2);
  return (p1 * a2 + p2 * a1) / (a1 + a2);
}
P isLL(L l1,L l2){ return isLL(l1[0],l1[1],l2[0],l2[1]); }
bool parallel(L l0, L l1) { return sign( l0.dir().det( l1.dir() ) ) ==
0; }
bool sameDir(L l0, L l1) { return parallel(l0, l1) && sign(l0.
dir().dot(l1.dir()) ) == 1; }
bool cmp (P a,  P b) {
  if (a.quad() != b.quad()) {
    return a.quad() < b.quad();
  } else {
    return sign( a.det(b) ) > 0;
  }
}
bool operator < (L l0, L l1) {
  if (sameDir(l0, l1)) {
    return l1.include(l0[0]);
  } else {
    return cmp( l0.dir(), l1.dir() );
  }
}
bool check(L u, L v, L w) {
  return w.include(isLL(u,v));
}
vector<P> halfPlaneIS(vector<L> &l) {
  sort(l.begin(), l.end());
  deque<L> q;
  for (int i = 0; i < (int)l.size(); ++i) {
    if (i && sameDir(l[i], l[i - 1])) continue;
    while (q.size() > 1 && !check(q[q.size() - 2], q[q.size() - 1],
l[i])) q.pop_back();
    while (q.size() > 1 && !check(q[1], q[0], l[i])) q.pop_front();
    q.push_back(l[i]);
  }
  while (q.size() > 2 && !check(q[q.size() - 2], q[q.size() - 1], q[0]))
q.pop_back();
  while (q.size() > 2 && !check(q[1], q[0], q[q.size() - 1]))
q.pop_front();
  vector<P> ret;
  for (int i = 0; i < (int)q.size(); ++i) ret.push_back(isLL(q[i], q[(i
+ 1) % q.size()]));
  return ret;
}
struct cmpX {
  bool operator()(P a, P b) const {
    return make_pair(a.x, a.y) < make_pair(b.x, b.y);
  }
};
bool intersect(double l1,double r1,double l2,double r2){
  if(l1>r1) swap(l1,r1); if(l2>r2) swap(l2,r2);
  return !( cmp(r1,l2) == -1 || cmp(r2,l1) == -1 );
}
bool isSS(P p1, P p2, P q1, P q2){
  return intersect(p1.x,p2.x,q1.x,q2.x) && intersect(p1.y,p2.y,q1.y,q2.
y) &&
  crossOp(p1,p2,q1) * crossOp(p1,p2,q2) <= 0 && crossOp(q1,q2,p1)
      * crossOp(q1,q2,p2) <= 0;
```

```
}
bool isSS_strict(P p1, P p2, P q1, P q2){
  return crossOp(p1,p2,q1) * crossOp(p1,p2,q2) < 0 && crossOp(q1,q2,p1)
      * crossOp(q1,q2,p2) < 0;
}
bool isMiddle(double a, double m, double b) {
  return sign(a - m) == 0 || sign(b - m) == 0 || (a < m != b < m);
}
bool isMiddle(P a, P m, P b) {
  return isMiddle(a.x, m.x, b.x) && isMiddle(a.y, m.y, b.y);
}
bool onSeg(P p1, P p2, P q){
  return crossOp(p1,p2,q) == 0 && isMiddle(p1, q, p2);
}
bool onSeg_strict(P p1, P p2, P q){
  return crossOp(p1,p2,q) == 0 && sign((q-p1).dot(p1-p2)) * sign((q-
p2).dot(p1-p2)) < 0;
}
double nearest(P p1,P p2,P q){
  P h = proj(p1,p2,q);
  if(isMiddle(p1,h,p2))
    return q.distTo(h);
  return min(p1.distTo(q),p2.distTo(q));
}
double disSS(P p1, P p2, P q1, P q2){
  if(isSS(p1,p2,q1,q2)) return 0;
  return min(min(nearest(p1,p2,q1),nearest(p1,p2,q2)), min(nearest(q1,
q2,p1),nearest(q1,q2,p2)));
}
double DEG_to_RAD(double d) { return d*M_PI/180.0; }
double RAD_to_DEG(double r) { return r*180.0/M_PI; }
double rad(P p1,P p2){
  return atan2l(p1.det(p2),p1.dot(p2));
}
bool inAngle(P a, P b, P c, P p) {
  assert(crossOp(a,b,c) != 0);
  if (crossOp(a,b,c) < 0) swap(b,c);
  return crossOp(a,b,p) >= 0 && crossOp(a,c,p) <= 0;
}
double angle(P v, P w) {
  return acos(clamp(v.dot(w) / v.abs() / w.abs(), -1.0, 1.0));
}
double orientedAngle(P a, P b, P c) { // BAC
  if (crossOp(a,b,c) >= 0) return angle(b-a, c-a);
  else return 2*M_PI - angle(b-a, c-a);
}
// double chord(double r, double ang) return sqrt(2*r*r*(1-
cos(ang))); // or 2*r*sin(ang/2)
// double secarea(double r, double ang) {return (ang/2)*(r*r);} // rad
// double segarea(double r, double ang) {return secarea(r, ang) -
r*r*sin(ang)/2;}
int type(P o1,double r1,P o2,double r2){
  double d = o1.distTo(o2);
  if(cmp(d,r1+r2) == 1) return 4; // outside each other
  if(cmp(d,r1+r2) == 0) return 3; // touch outside
  if(cmp(d,abs(r1-r2)) == 1) return 2; // one inside another
  if(cmp(d,abs(r1-r2)) == 0) return 1; // touch inside
  return 0;
}
vector<P> isCL(P o,double r,P p1,P p2){
  if (cmp(abs((o-p1).det(p2-p1)/p1.distTo(p2)),r)>0) return {};
  double x = (p1-o).dot(p2-p1), y = (p2-p1).abs2(), d = x * x - y *
((p1-o).abs2() - r*r);
  d = max(d,0.0); P m = p1 - (p2-p1)*(x/y), dr = (p2-p1)*(sqrt(d)/y);
  return {m-dr,m+dr}; //along dir: p1->p2
}
vector<P> isCC(P o1, double r1, P o2, double r2) { //need to check
whether two circles are the same
  double d = o1.distTo(o2);
  if (cmp(d, r1 + r2) == 1) return {};
  if (cmp(d,abs(r1-r2))==-1) return {};
  d = min(d, r1 + r2);
  double y = (r1 * r1 + d * d - r2 * r2) / (2 * d), x = sqrt(r1 * r1 - y
* y);
  P dr = (o2 - o1).unit();
  P q1 = o1 + dr * y, q2 = dr.rot90() * x;
  return {q1-q2,q1+q2};//along circle 1
}
vector<P> tanCP(P o, double r, P p) {
  double x = (p - o).abs2(), d = x - r * r;
  if (sign(d) <= 0) return {}; // on circle => no tangent
  P q1 = o + (p - o) * (r * r / x);
  P q2 = (p - o).rot90() * (r * sqrt(d) / x);
  return {q1-q2,q1+q2}; //counter clock-wise
}
vector<L> extanCC(P o1, double r1, P o2, double r2) {
  vector<L> ret;
  if (cmp(r1, r2) == 0) {
    P dr = (o2 - o1).unit().rot90() * r1;
    ret.push_back(L(o1 + dr, o2 + dr)), ret.push_back(L(o1 - dr, o2 -
dr));
  } else {
    P p = (o2 * r1 - o1 * r2) / (r1 - r2);
    vector<P> ps = tanCP(o1, r1, p), qs = tanCP(o2, r2, p);
    for(int i = 0; i < min(ps.size(),qs.size());i++)
ret.push_back(L(ps[i], qs[i])); //c1 counter-clock wise
  }
  return ret;
}
vector<L> intanCC(P o1, double r1, P o2, double r2) {
  vector<L> ret;
  P p = (o1 * r2 + o2 * r1) / (r1 + r2);
  vector<P> ps = tanCP(o1,r1,p), qs = tanCP(o2,r2,p);
  for(int i = 0; i < min(ps.size(),qs.size()); i++)
```

```
ret.push_back(L(ps[i], qs[i])); //c1 counter-clock wise
  return ret;
}
double areaCT(double r, P p1, P p2){
  vector<P> is = isCL(P(0,0),r,p1,p2);
  if(is.empty()) return r*r*rad(p1,p2)/2;
  bool b1 = cmp(p1.abs2(),r*r) == 1, b2 = cmp(p2.abs2(), r*r) == 1;
  if(b1 && b2){
    if(sign((p1-is[0]).dot(p2-is[0])) <= 0 &&
      sign((p1-is[0]).dot(p2-is[0])) <= 0)
    return r*r*(rad(p1,is[0]) + rad(is[1],p2))/2 + is[0].det(is[1])/2;
    else return r*r*rad(p1,p2)/2;
  }
  if(b1) return (r*r*rad(p1,is[0]) + is[0].det(p2))/2;
  if(b2) return (p1.det(is[1]) + r*r*rad(is[1],p2))/2;
  return p1.det(p2)/2;
}
P inCenter(P A, P B, P C) {
  double a = (B - C).abs(), b = (C - A).abs(), c = (A - B).abs();
  return (A * a + B * b + C * c) / (a + b + c);
}
P circumCenter(P a, P b, P c) {
  P bb = b - a, cc = c - a;
  double db = bb.abs2(), dc = cc.abs2(), d = 2 * bb.det(cc);
  return a - P(bb.y * dc - cc.y * db, cc.x * db - bb.x * dc) / d;
}
P othroCenter(P a, P b, P c) {
  P ba = b - a, ca = c - a, bc = b - c;
  double Y = ba.y * ca.y * bc.y,
  A = ca.x * ba.y - ba.x * ca.y,
  x0 = (Y + ca.x * ba.y * b.x - ba.x * ca.y * c.x) / A,
  y0 = -ba.x * (x0 - c.x) / ba.y + ca.y;
  return {x0, y0};
}
//polygon
double area(vector<P> ps){
  double ret = 0;   for(int i=0; i< ps.size(); i++) ret +=
ps[i].det(ps[(i+1)%ps.size()]);
  return ret/2;
}
int contain(vector<P> ps, P p){ //2:inside,1:on_seg,0:outside
  int n = ps.size(), ret = 0;   for(int i = 0; i < n; i++) {
    P u=ps[i],v=ps[(i+1)%n];
    if(onSeg(u,v,p)) return 1;
    if(cmp(u.y,v.y)<=0) swap(u,v);
    if(cmp(p.y,u.y) >0 || cmp(p.y,v.y) <= 0) continue;
    ret ^= crossOp(p,u,v) > 0;
  }
  return ret*2;
}
vector<P> convexHull(vector<P> ps) {
  int n = ps.size(); if(n <= 1) return ps;
  sort(ps.begin(), ps.end());
  vector<P> qs(n * 2); int k = 0;
  for (int i = 0; i < n; qs[k++] = ps[i++])
    while (k > 1 && crossOp(qs[k - 2], qs[k - 1], ps[i]) <= 0) --k;
  for (int i = n - 2, t = k; i >= 0; qs[k++] = ps[i--])
    while (k > t && crossOp(qs[k - 2], qs[k - 1], ps[i]) <= 0) --k;
  qs.resize(k - 1);
  return qs;
}
vector<P> convexHullNonStrict(vector<P> ps) {
  //caution: need to unique the Ps first
  int n = ps.size(); if(n <= 1) return ps;
  sort(ps.begin(), ps.end());
  vector<P> qs(n * 2); int k = 0;
  for (int i = 0; i < n; qs[k++] = ps[i++])
    while (k > 1 && crossOp(qs[k - 2], qs[k - 1], ps[i]) < 0) --k;
  for (int i = n - 2, t = k; i >= 0; qs[k++] = ps[i--])
    while (k > t && crossOp(qs[k - 2], qs[k - 1], ps[i]) < 0) --k;
  qs.resize(k - 1);
  return qs;
}
double convexDiameter(vector<P> ps){
  int n = ps.size(); if(n <= 1) return 0;
  int is = 0, js = 0; for(int k = 1; k < n; k++) is = ps[k]<ps[is]?k:is,
js = ps[js] < ps[k]?k:js;
  int i = is, j = js;
  double ret = ps[i].distTo(ps[j]);
  do{
    if((ps[(i+1)%n]-ps[i]).det(ps[(j+1)%n]-ps[j]) >= 0)
      (++j)%=n;
    else
      (++i)%=n;
    ret = max(ret,ps[i].distTo(ps[j]));
  }while(i!=is || j!=js);
  return ret;
}
vector<P> convexCut(const vector<P>&ps, P q1, P q2) {
  vector<P> qs;
  int n = ps.size();
  for(int i = 0; i<n; i++) {
    P p1 = ps[i], p2 = ps[(i+1)%n];
    int d1 = crossOp(q1,q2,p1), d2 = crossOp(q1,q2,p2);
    if(d1 >= 0) qs.push_back(p1);
    if(d1 * d2 < 0) qs.push_back(isLL(p1,p2,q1,q2));
  }
  return qs;
}
```

Geometry/line.h

```
struct L {
  P ps[2]; P v; T c;
  L() {}
  P& operator[](int i) { return ps[i]; }
  // From direction vector v and offset c
  L(P v, T c) : v(v), c(c) {}
  // From equation ax+by=c
  L(T a, T b, T c) : v({b,-a}), c(c) {}
  // From points P and Q
  L(P p, P q) : v(q-p), c(cross(P(0, 0), v,p)) {
    ps[0] = p;
    ps[1] = q;
  }
  P dir() { return ps[1] - ps[0]; }
  bool include(P p) { return sign((ps[1] - ps[0]).det(p - ps[0])) > 0; }
  T side(P p) {return cross(P(0, 0), v,p)-c;}
  T dist(P p) {return abs(side(p)) / v.abs();}
  T sqDist(P p) {return side(p)*side(p) / (double)v.abs();}
  L perpThrough(P p) {return L(p, p + v.rot90());}
  bool cmpProj(P p, P q) {
    return v.dot(p) < v.dot(q);
  }
  L translate(P t) {return L(v, c + cross(P(0,0), v,t));}
  L shiftLeft(double dist) {return L(v, c + dist*v.abs());}
  L shiftRight(double dist) {return L(v, c - dist*v.abs());}
};
bool chkLL(P p1, P p2, P q1, P q2) {
  double a1 = cross(q1, q2, p1), a2 = -cross(q1, q2, p2);
  return sign(a1+a2) != 0;
}
P isLL(P p1, P p2, P q1, P q2) {
  double a1 = cross(q1, q2, p1), a2 = -cross(q1, q2, p2);
  return (p1 * a2 + p2 * a1) / (a1 + a2);
}
P isLL(L l1,L l2){ return isLL(l1[0],l1[1],l2[0],l2[1]); }
bool parallel(L l0, L l1) { return sign( l0.dir().det( l1.dir() ) ) ==
0; }
bool sameDir(L l0, L l1) { return parallel(l0, l1) && sign(l0.
dir().dot(l1.dir()) ) == 1; }
bool cmp (P a,  P b) {
  if (a.quad() != b.quad()) {
    return a.quad() < b.quad();
  } else {
    return sign( a.det(b) ) > 0;
  }
}
bool operator < (L l0, L l1) {
  if (sameDir(l0, l1)) {
    return l1.include(l0[0]);
  } else {
    return cmp( l0.dir(), l1.dir() );
  }
}
bool check(L u, L v, L w) {
  return w.include(isLL(u,v));
}
vector<P> halfPlaneIS(vector<L> &l) {
  sort(l.begin(), l.end());
  deque<L> q;
  for (int i = 0; i < (int)l.size(); ++i) {
    if (i && sameDir(l[i], l[i - 1])) continue;
    while (q.size() > 1 && !check(q[q.size() - 2], q[q.size() - 1],
l[i])) q.pop_back();
    while (q.size() > 1 && !check(q[1], q[0], l[i])) q.pop_front();
    q.push_back(l[i]);
  }
  while (q.size() > 2 && !check(q[q.size() - 2], q[q.size() - 1], q[0]))
q.pop_back();
  while (q.size() > 2 && !check(q[1], q[0], q[q.size() - 1]))
q.pop_front();
  vector<P> ret;
  for (int i = 0; i < (int)q.size(); ++i) ret.push_back(isLL(q[i], q[(i
+ 1) % q.size()]));
  return ret;
}
```

Geometry/point.h

```
typedef double T;
const double EPS = 1e-9;
inline int sign(double a) { return a < -EPS ? -1 : a > EPS; }
inline int cmp(double a, double b){ return sign(a-b); }
struct P {
  T x,y;
  P() {}
  P(T _x, T _y) : x(_x), y(_y) {}
  P operator+(P p) {return {x+p.x, y+p.y};}
  P operator-(P p) {return {x-p.x, y-p.y};}
  P operator*(T d) {return {x*d, y*d};}
  P operator/(T d) {return {x/d, y/d};} // only for floatingpoint
  bool operator<(P p) const {
    int c = cmp(x, p.x);
    if (c) return c == -1;
    return cmp(y, p.y) == -1;
  }
  bool operator==(P o) const{
    return cmp(x,o.x) == 0 && cmp(y,o.y) == 0;
  }
  double dot(P p) { return x * p.x + y * p.y; }
  double det(P p) { return x * p.y - y * p.x; }
  double distTo(P p) { return (*this-p).abs(); }
  double alpha() { return atan2(y, x); }
  void read() { cin>>x>>y; }
  void write() {cout<<"("<<x<<","<<y<<")"<<endl;}
  double abs() { return sqrt(abs2());}
  double abs2() { return x * x + y * y; }
  P rot90() { return P(-y,x);}
  P unit() { return *this/abs(); }
  int quad() const { return sign(y) == 1 || (sign(y) == 0 && sign(x) >=
0); }
  P rot(double an){ return {x*cos(an)-y*sin(an),x*sin(an) +
y*cos(an)}; }
};
#define cross(p1,p2,p3) ((p2.x-p1.x)*(p3.y-p1.y)-(p3.x-p1.x)*(p2.y-p1.
y))
#define crossOp(p1,p2,p3) sign(cross(p1,p2,p3))
bool isConvex(vector<P> p) {
  bool hasPos=false, hasNeg=false;
  for (int i=0, n=p.size(); i<n; i++) {
    int o = cross(p[i], p[(i+1)%n], p[(i+2)%n]);
    if (o > 0) hasPos = true;
    if (o < 0) hasNeg = true;
  }
  return !(hasPos && hasNeg);
}
bool half(P p) {
  assert(p.x != 0 || p.y != 0); // (0, 0) is not covered
  return p.y > 0 || (p.y == 0 && p.x < 0);
}
void polarSortAround(P o, vector<P> &v) {
  sort(v.begin(), v.end(), [&o](P v, P w) {
      return make_tuple(half(v-o), 0) <
        make_tuple(half(w-o), cross(o, v, w));
  });
}
P proj(P p1, P p2, P q) {
  P dir = p2 - p1;
  return p1 + dir * (dir.dot(q - p1) / dir.abs2());
}
P reflect(P p1, P p2, P q){
  return proj(p1,p2,q) * 2 - q;
}
// tested with https://open.kattis.com/problems/closestpair2
pair<P, P> closest(vector<P> v) {
  assert(sz(v) > 1);
  set <P> S;
  sort(v.begin(), v.end(), [](P a, P b) { return a.y < b.y; });
  pair<T, pair<P, P>> ret{(T)1e18, {P(), P()}};
  int j = 0;
  for(P p : v) {
    P d { 1 + (T) sqrt(ret.first), 0 };
    while(p.y - v[j].y >= d.x) S.erase(v[j++]);
    auto lo = S.lower_bound(p - d), hi = S.upper_bound(p + d);
    for(; lo != hi; ++lo) {
      ret = min(ret, {(p - (*lo)).abs2(), {*lo, p}});
    }
    S.insert(p);
  }
  return ret.second;
}
```

Geometry/polygon.h

```
//polygon
double area(vector<P> ps){
  double ret = 0;   for(int i=0; i< ps.size(); i++) ret +=
ps[i].det(ps[(i+1)%ps.size()]);
  return ret/2;
}
int contain(vector<P> ps, P p){ //2:inside,1:on_seg,0:outside
  int n = ps.size(), ret = 0;   for(int i = 0; i < n; i++) {
    P u=ps[i],v=ps[(i+1)%n];
    if(onSeg(u,v,p)) return 1;
    if(cmp(u.y,v.y)<=0) swap(u,v);
    if(cmp(p.y,u.y) >0 || cmp(p.y,v.y) <= 0) continue;
    ret ^= crossOp(p,u,v) > 0;
  }
  return ret*2;
}
vector<P> convexHull(vector<P> ps) {
  int n = ps.size(); if(n <= 1) return ps;
  sort(ps.begin(), ps.end());
  vector<P> qs(n * 2); int k = 0;
  for (int i = 0; i < n; qs[k++] = ps[i++])
    while (k > 1 && crossOp(qs[k - 2], qs[k - 1], ps[i]) <= 0) --k;
  for (int i = n - 2, t = k; i >= 0; qs[k++] = ps[i--])
    while (k > t && crossOp(qs[k - 2], qs[k - 1], ps[i]) <= 0) --k;
  qs.resize(k - 1);
  return qs;
}
vector<P> convexHullNonStrict(vector<P> ps) {
  //caution: need to unique the Ps first
  int n = ps.size(); if(n <= 1) return ps;
  sort(ps.begin(), ps.end());
  vector<P> qs(n * 2); int k = 0;
  for (int i = 0; i < n; qs[k++] = ps[i++])
    while (k > 1 && crossOp(qs[k - 2], qs[k - 1], ps[i]) < 0) --k;
  for (int i = n - 2, t = k; i >= 0; qs[k++] = ps[i--])
    while (k > t && crossOp(qs[k - 2], qs[k - 1], ps[i]) < 0) --k;
  qs.resize(k - 1);
  return qs;
}
double convexDiameter(vector<P> ps){
  int n = ps.size(); if(n <= 1) return 0;
  int is = 0, js = 0; for(int k = 1; k < n; k++) is = ps[k]<ps[is]?k:is,
js = ps[js] < ps[k]?k:js;
  int i = is, j = js;
  double ret = ps[i].distTo(ps[j]);
  do{
    if((ps[(i+1)%n]-ps[i]).det(ps[(j+1)%n]-ps[j]) >= 0)
```

```
      (++j)%=n;
    else
      (++i)%=n;
    ret = max(ret,ps[i].distTo(ps[j]));
  }while(i!=is || j!=js);
  return ret;
}
vector<P> convexCut(const vector<P>&ps, P q1, P q2) {
  vector<P> qs;
  int n = ps.size();
  for(int i = 0; i<n; i++) {
    P p1 = ps[i], p2 = ps[(i+1)%n];
    int d1 = crossOp(q1,q2,p1), d2 = crossOp(q1,q2,p2);
    if(d1 >= 0) qs.push_back(p1);
    if(d1 * d2 < 0) qs.push_back(isLL(p1,p2,q1,q2));
  }
  return qs;
}
```

Geometry/segment.h

```
struct cmpX {
  bool operator()(P a, P b) const {
    return make_pair(a.x, a.y) < make_pair(b.x, b.y);
  }
};
bool intersect(double l1,double r1,double l2,double r2){
  if(l1>r1) swap(l1,r1); if(l2>r2) swap(l2,r2);
  return !( cmp(r1,l2) == -1 || cmp(r2,l1) == -1 );
}
bool isSS(P p1, P p2, P q1, P q2){
  return intersect(p1.x,p2.x,q1.x,q2.x) && intersect(p1.y,p2.y,q1.y,q2.
y) &&
  crossOp(p1,p2,q1) * crossOp(p1,p2,q2) <= 0 && crossOp(q1,q2,p1)
      * crossOp(q1,q2,p2) <= 0;
}
bool isSS_strict(P p1, P p2, P q1, P q2){
  return crossOp(p1,p2,q1) * crossOp(p1,p2,q2) < 0 && crossOp(q1,q2,p1)
      * crossOp(q1,q2,p2) < 0;
}
bool isMiddle(double a, double m, double b) {
  return sign(a - m) == 0 || sign(b - m) == 0 || (a < m != b < m);
}
bool isMiddle(P a, P m, P b) {
  return isMiddle(a.x, m.x, b.x) && isMiddle(a.y, m.y, b.y);
}
bool onSeg(P p1, P p2, P q){
  return crossOp(p1,p2,q) == 0 && isMiddle(p1, q, p2);
}
bool onSeg_strict(P p1, P p2, P q){
  return crossOp(p1,p2,q) == 0 && sign((q-p1).dot(p1-p2)) * sign((q-
p2).dot(p1-p2)) < 0;
}
double nearest(P p1,P p2,P q){
  P h = proj(p1,p2,q);
  if(isMiddle(p1,h,p2))
    return q.distTo(h);
  return min(p1.distTo(q),p2.distTo(q));
}
double disSS(P p1, P p2, P q1, P q2){
  if(isSS(p1,p2,q1,q2)) return 0;
  return min(min(nearest(p1,p2,q1),nearest(p1,p2,q2)), min(nearest(q1,
q2,p1),nearest(q1,q2,p2)));
}
```

Graph/2-sat.cpp

```
// suppose you have some boolean variables a, b, c, d...
// assign each variable true or false such that the expression like
// the following is true:
// (a or not b) and (not a or b) and (not a or not b) and (a or not c)
// the expression is a conjunction of multiple clauses, where each
// clause is a disjunction of exactly two literals

#include <vector>

struct TwoSAT {
    int n;
    std::vector<std::vector<int>> g;
    TwoSAT(int n) : n(n), g(n * 2) {} // n is the number of literals
                                      // 2 * u represents the node u
                                      // 2 * u + 1 represents the node !
u

    void add(int u, bool neg_u, int v, bool neg_v) { // neg_u is if u is
negated, same for v
        g[2 * u + !neg_u].push_back(2 * v + neg_v);
        g[2 * v + !neg_v].push_back(2 * u + neg_u);
    }

    std::vector<bool> solve() {
        auto [cnt, color] = scc(g);

        std::vector<bool> res(n);
        for (int i = 0; i < n; i++) {
            if (color[2 * i] == color[2 * i + 1]) return {};
            // as Tarjan's algorithm finds node in reverse topological
order,
            // color[2 * i] < color[2 * i + 1] => there might be a path
fron !i to i
            // so it's safe to set i = true
            res[i] = color[2 * i] < color[2 * i + 1];
        }
        return res;
    }
};
```

Graph/BellmanFord.cpp

```
struct BellmanFord {
    static constexpr long long INF=1e18;
    int n, last_relaxed=-1;
    vector<tuple<int, int, int>> edges;
    vector<bool> bad; //has negative cycle on the path
    vector<int> pre;
    vector<ll> dis;
    BellmanFord(int _n) : n(_n), bad(n), pre(n), dis(n, INF) {}
    void add_edge(int u, int v, int w) {
        edges.emplace_back(u, v, w);
    }
    void run(int start) {
        dis[start]=0;
        for (int i=0; i<n-1; i++) {
            for (auto [u, v, w] : edges) {
                if (dis[u]<INF && dis[v]>dis[u]+w) {
                    dis[v]=dis[u]+w;
                    pre[v]=u;
                }
            }
        }
        for (auto [u, v, w] : edges) {
            if (dis[u]<INF && dis[v]>dis[u]+w) {
                dis[v]=dis[u]+w;
                bad[v]=true;
                last_relaxed=v;
                pre[v]=u;
            }
        }
        for (int i=0; i<n; i++) {
            for (auto [u, v, w] : edges) {
                if (bad[u]) bad[v]=true;
            }
        }
    }
    vector<int> find_cycle() {
        dis.assign(n, 0); // without this, only cycle reachable from 0
will be counted
        run(0);
        if (last_relaxed==-1) return {};
        int x=last_relaxed;
        for (int i=0; i<n; i++) x=pre[x];
        vector<int> cycle;
        for (int cur=x; ; cur=pre[cur]) {
            cycle.push_back(cur);
            if (cur==x && cycle.size()>1) break;
        }
        reverse(cycle.begin(), cycle.end());
        return cycle;
    }
    long long get_dis(int x) {
        return bad[x] ? -INF : dis[x];
    }
};
```

Graph/MCMF.cpp

```
struct Flow {
  static inline constexpr ll INF = INT64_MAX >> 1;
  int n;
  vector<tuple<int, int, int>> e;
  vector<vector<int>> g;
  vector<int> prev;
  vector<ll> h; // distance, also potential
  Flow(int n) : n(n), g(n), h(n), prev(n) {}
  void addEdge(int u, int v, int w, int c) {
    if (u == v) return;
    g[u].emplace_back(e.size());
    e.emplace_back(v, w, c);
    g[v].emplace_back(e.size());
    e.emplace_back(u, 0, -c);
  }
  bool dijkstra(int s, int t) {
    priority_queue<pair<ll, int>> q;
    fill(prev.begin(), prev.end(), -1);
    vector<ll> d(n, INF);
    d[s] = 0;
    q.push({0, s});
    while (!q.empty()) {
      auto [du, u] = q.top();
      q.pop();
      if (d[u] != -du) continue;
      for (auto i : g[u]) {
        auto [v, w, c] = e[i];
        c += h[u] - h[v];
        if (w > 0 && d[v] > d[u] + c) {
          d[v] = d[u] + c;
          prev[v] = i;
          q.push({-d[v], v});
        }
      }
    }
    for (int i = 0; i < n; ++i) {
      if ((h[i] += d[i]) > INF) h[i] = INF;
    }
    return h[t] != INF;
  }
  pair<ll, ll> maxFlow(int s, int t) {
    ll flow = 0, cost = 0;
    while (dijkstra(s, t)) {
      int f = INT_MAX, now = t;
      vector<int> r;
      while (now != s) {
        r.emplace_back(prev[now]);
        f = min(f, get<1>(e[prev[now]]));
        now = get<0>(e[prev[now] ^ 1]);
      }
      for (auto i : r) {
        get<1>(e[i]) -= f;
        get<1>(e[i ^ 1]) += f;
      }
      flow += f;
      cost += ll(f) * h[t];
    }
    return {flow, cost};
  }
};
```

Graph/augmented_path_BPM.cpp

```
#include <bits/stdc++.h>
using namespace std;
// augmented path algorithm for maximum-caredinality bipartite matching
// Worst time complexity: O(nm), but very hard to hack (since we can
shuffle),
// usually runs extremely fast, 2e5 vertices and edges in 60 ms.

int main() {
    ios::sync_with_stdio(false);
    int l, r, m;
    cin >> l >> r >> m;
    vector<vector<int>> g(l);
    while (m--) {
        int u, v;
        cin >> u >> v;
        u--, v--;
        g[u].push_back(v); // note how we build the graph
    }
    auto aug_path = [](int n, int m, const vector<vector<int>> &g) {
        // we can shuffle vertices:
        // for (auto& v : g)
        //     shuffle(v.begin(), v.end(), rng);
        vector<int> l(n, -1), r(m, -1), vis(n);
        auto match = [&](auto& slf, int u) {
            if (vis[u]) return false;
            vis[u] = true;
            for (auto v : g[u]) {
                if (r[v] == -1) {
                    l[u] = v;
                    r[v] = u;
                    return true;
                }
            }
            for (auto v : g[u]) {
                if (slf(slf, r[v])) {
                    l[u] = v;
                    r[v] = u;
                    return true;
                }
            }
            return false;
        };

        // We can also shuffle the order of visiting vertices
        // vector<int> order(L.size());
        // iota(order.begin(), order.end(), 0);
        // shuffle(order.begin(), order.end(), rng);
        bool ok = true;
        while (ok) {
            ok = false;
            fill(vis.begin(), vis.end(), 0);
            for (int i = 0; i < n; ++i) {
                if (l[i] == -1) ok |= match(match, i);
            }
        }

        std::vector<std::pair<int, int>> matches;
        for (size_t i = 0; i < n; i++) {
            if (l[i] != -1) {
                matches.emplace_back(i, l[i]);
            }
        }
        return matches;
    };
    auto res = aug_path(l, r, g);
}
```

Graph/biconnected_components.cpp

```
#include <vector>
using namespace std;
int main() {
    auto biconnected_comp = [&](const vector<vector<int>> &g) {
        const int n = (int)size(g);
        int pos = 0;
        vector<int> ord(n, -1), low(n), cuts, stk;
        vector<vector<int>> comps; // components

        auto dfs = [&](auto &slf, int u, int pa) -> void {
            low[u] = ord[u] = pos++;
            stk.push_back(u);
            int cnt = 0;
            bool is_cut = false;
            for (auto v : g[u]) {
                if (v == pa)
                    continue;
                if (ord[v] == -1) {
                    cnt++;
                    slf(slf, v, u);
                    low[u] = min(low[u], low[v]);
                    if (low[v] >= ord[u]) {
                        if (u != pa || cnt > 1)
                            is_cut = true;
                        // the subtree will be disconnected if we remove
                        // vertex u, do something if needed
                        comps.emplace_back();
                        while (true) {
                            int back = stk.back();
                            stk.pop_back();
                            comps.back().push_back(back);
                            if (back == v)
                                break;
                        }
                        comps.back().push_back(u);
                    }
                } else {
                    low[u] = min(low[u], ord[v]);
                }
            }
            if (is_cut)
                cuts.push_back(u);
        };

        for (int i = 0; i < n; i++) {
            if (ord[i] == -1)
                dfs(dfs, i, i);
        }

        return comps;
    };
}
/*
 * Extension: round-square tree
 * Let c be the number of biconnected components in a graph G. The
round-square
 * tree consists of n round vertices and c square vertices. Each round
vertex is
 * connected to the square vertices corresponding to the biconnected
components
 * that the round vertex belongs to.
 * The round-square tree is a tree with n + c vertices and n + c − 1
edges.
 *
 * Example (ABC 318G):
 * Given a graph and three vertices A, B, C. Determine if there is a
simple path
 * connecting vertices A and C via vertex B.
 *
 * Solution:
 * In the round-square tree, check if there is a square vertex on the
path from
 * A to C that is connected to B by an edge.
 */
```

Graph/binary_lifting.cpp

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

int __lg(int);
int op(int, int);
int e();
int main() {
    int n;
    vector<vector<array<int, 2>>> g(n);

    const int lg = __lg(n);
    vector pa(n, vector(lg + 1, 0));
    vector data(n, vector(lg + 1, 0)); // data[u][i]: data of path from
u to pa[u][i]
    data[0][0] = e();                  // set data[root][0] to identity
element
    vector<int> dep(n);

    auto dfs = [&](auto& slf, int u, int p) -> void {
        pa[u][0] = p;
        for (int i = 1; i <= lg; i++) {
            pa[u][i] = pa[pa[u][i - 1]][i - 1];
            data[u][i] = op(data[u][i - 1], data[pa[u][i - 1]][i - 1]);
        }

        for (auto [v, w] : g[u]) {
            if (v == p) continue;
            data[v][0] = w;
            dep[v] = dep[u] + 1;
            slf(slf, v, u);
        }
    };

    dfs(dfs, 0, 0);

    auto jump = [&](int u, int d) {
        auto s = e();
        for (int i = lg; i >= 0; i--) {
            if (d >> i & 1) {
                s = op(s, data[u][i]);
                u = pa[u][i];
            }
        }
        return pair{u, s};
    };

    auto lca = [&](int u, int v) {
        if (dep[u] < dep[v]) {
            swap(u, v);
        }
        int s = e();
        tie(u, s) = jump(u, dep[u] - dep[v]);

        if (u == v) return pair{u, s};

        for (int i = lg; i >= 0; i--) {
            if (pa[u][i] != pa[v][i]) {
                s = op(op(s, data[u][i]), data[v][i]);
                u = pa[u][i];
                v = pa[v][i];
            }
        }

        s = op(op(s, data[u][0]), data[v][0]);
        return pair{pa[u][0], s};
    };
}
```

Graph/blossom.cpp

```
// https://codeforces.com/blog/entry/92339
// another faster algorithm https://judge.yosupo.jp/submission/51928
#include <bits/stdc++.h>

using namespace std;

struct blossom {
    int n, m;
    vector<int> mate;
    vector<vector<int>> b;
    vector<int> p, d, bl;
    vector<vector<int>> g;
    blossom(int n) : n(n) {
        m = n + n / 2;
        mate.assign(n, -1);
        b.resize(m);
        p.resize(m);
        d.resize(m);
        bl.resize(m);
        g.assign(m, vector<int>(m, -1));
    }
    void add_edge(int u, int v) {
        g[u][v] = u;
        g[v][u] = v;
    }
    void match(int u, int v) {
        g[u][v] = g[v][u] = -1;
        mate[u] = v;
        mate[v] = u;
    }
    vector<int> trace(int x) {
        vector<int> vx;
        while(true) {
            while(bl[x] != x) x = bl[x];
            if(!vx.empty() && vx.back() == x) break;
            vx.push_back(x);
            x = p[x];
        }
        return vx;
    }
    void contract(int c, int x, int y, vector<int> &vx, vector<int> &vy)
{
        b[c].clear();
        int r = vx.back();
        while(!vx.empty() && !vy.empty() && vx.back() == vy.back()) {
            r = vx.back();
            vx.pop_back();
            vy.pop_back();
        }
        b[c].push_back(r);
        b[c].insert(b[c].end(), vx.rbegin(), vx.rend());
        b[c].insert(b[c].end(), vy.begin(), vy.end());
        for(int i = 0; i <= c; i++) {
            g[c][i] = g[i][c] = -1;
        }
        for(int z : b[c]) {
            bl[z] = c;
            for(int i = 0; i < c; i++) {
                if(g[z][i] != -1) {
                    g[c][i] = z;
                    g[i][c] = g[i][z];
                }
            }
        }
    }
    vector<int> lift(vector<int> &vx) {
        vector<int> A;
        while(vx.size() >= 2) {
            int z = vx.back(); vx.pop_back();
            if(z < n) {
                A.push_back(z);
                continue;
            }
            int w = vx.back();
            int i = (A.size() % 2 == 0 ? find(b[z].begin(), b[z].end(),
g[z][w]) - b[z].begin() : 0);
            int j = (A.size() % 2 == 1 ? find(b[z].begin(), b[z].end(),
g[z][A.back()]) - b[z].begin() : 0);
            int k = b[z].size();
            int dif = (A.size() % 2 == 0 ? i % 2 == 1 : j % 2 == 0) ?
1 : k - 1;
            while(i != j) {
                vx.push_back(b[z][i]);
                i = (i + dif) % k;
            }
            vx.push_back(b[z][i]);
        }
        return A;
    }
    int solve() {
        for(int ans = 0; ; ans++) {
            fill(d.begin(), d.end(), 0);
            queue<int> Q;
            for(int i = 0; i < m; i++) bl[i] = i;
            for(int i = 0; i < n; i++) {
                if(mate[i] == -1) {
                    Q.push(i);
                    p[i] = i;
                    d[i] = 1;
                }
            }
            int c = n;
            bool aug = false;
            while(!Q.empty() && !aug) {
                int x = Q.front(); Q.pop();
                if(bl[x] != x) continue;
                for(int y = 0; y < c; y++) {
                    if(bl[y] == y && g[x][y] != -1) {
                        if(d[y] == 0) {
                            p[y] = x;
                            d[y] = 2;
                            p[mate[y]] = y;
                            d[mate[y]] = 1;
                            Q.push(mate[y]);
                        }else if(d[y] == 1) {
                            vector<int> vx = trace(x);
                            vector<int> vy = trace(y);
                            if(vx.back() == vy.back()) {
                                contract(c, x, y, vx, vy);
                                Q.push(c);
                                p[c] = p[b[c][0]];
                                d[c] = 1;
                                c++;
                            }else {
                                aug = true;
                                vx.insert(vx.begin(), y);
                                vy.insert(vy.begin(), x);
```

```
                                vector<int> A = lift(vx);
                                vector<int> B = lift(vy);
                                A.insert(A.end(), B.rbegin(), B.rend());
                                for(int i = 0; i < (int) A.size(); i +=
2) {
                                    match(A[i], A[i + 1]);
                                    if(i + 2 < (int) A.size())
add_edge(A[i + 1], A[i + 2]);
                                }
                            }
                            break;
                        }
                    }
                }
            }
            if(!aug) return ans;
        }
    }
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    int n, m;
    cin >> n >> m;
    blossom B(n);
    for(int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        B.add_edge(u, v);
    }
    cout << B.solve() << '\n';
    for(int i = 0; i < n; i++) {
        if(i < B.mate[i]) {
            cout << i << ' ' << B.mate[i] << '\n';
        }
    }
};
```

Graph/bridges.cpp

```
struct Bridge {
    int n, pos=0;
    vector<vector<pair<int, int>>> g; // graph, component
    vector<int> ord, low, bridges; // order, low link, belong to which
component
    Bridge(int n) : n(n), g(n), ord(n, -1), low(n) {}
    void add_edge(int u, int v, int i) {
        g[u].emplace_back(v, i);
        g[v].emplace_back(u, i);
    }
    void dfs(int u, int p) {
        ord[u] = low[u] = pos++;
        int cnt = 0;
        for (auto [v, i] : g[u]) {
            // in case there're repeated edges, only skip the first one
            if (v == p && cnt == 0) {
                cnt++;
                continue;
            }
            if (ord[v] == -1) dfs(v, u);
            low[u] = min(low[u], low[v]);
            if (low[v] > ord[u]) bridges.push_back(i);
        }
    }
    void solve() {
        for (int i = 0; i < n; i++)
            if (ord[i] == -1) dfs(i, i);
    }
};
```

Graph/centroid-decomposition.cpp

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

int main() {
    int n;
    vector<vector<array<int, 2>>> g;

    auto decomp = [&](auto& f) {
        vector<int> vis(n), sz(n, 1);
        auto cal_sz = [&](auto& slf, int u, int p) -> void {
            for (auto [v, w] : g[u]) {
                if (v == p) continue;
                slf(slf, v, u);
                sz[u] += sz[v];
            }
        };
        cal_sz(cal_sz, 0, 0);

        vector<vector<array<int, 2>>> tr(n);
        auto go = [&](auto& slf, int u) -> void {
            int s = sz[u];
            int prev = -1;
            while (1) {
                for (auto [v, w] : g[u]) {
                    if (!vis[v] && sz[v] * 2 > s) {
                        sz[u] -= sz[v];
                        sz[v] = s;
                        u = v;
                    }
                }
                if (u == prev) break;
                prev = u;
            }

            vis[u] = 1;
            for (auto [v, w] : g[u]) {
                if (!vis[v]) {
                    slf(slf, v);
                    tr[u].push_back({v, w});
                    tr[v].push_back({u, w});
                }
            }
            f(tr, u, s); // u is the root of the current tree, s is the
size of the tree
        };
        go(go, 0);
    };
}
```

Graph/count-cycles.cpp

```
#include <vector>
#include <algorithm>
#include <numeric>

// $O(m \sqrt{m})$, we will get TLE if the answer greater than INT_MAX
static int circle3count(const std::vector<std::pair<int, int>>& edge,
int n) {
  std::vector<int> d(n), vis(n, -1);
  for (auto [u, v] : edge) ++d[u], ++d[v];
  std::vector<std::vector<int>> e(n);
  // Giving Orienting to Edge
  for (auto [u, v] : edge) {
    if (d[u] < d[v] || (d[u] == d[v] && u < v)) {
      e[u].emplace_back(v);
    } else {
      e[v].emplace_back(u);
    }
  }
  int ans = 0;
  for (int i = 0; i < n; ++i) {
    for (auto u : e[i]) vis[u] = i;
    for (auto u : e[i]) {
      for (auto v : e[u]) if (vis[v] == i) ++ans;
    }
  }
  return ans;
}
// https://www.luogu.com.cn/problem/P1989


// $O(m \sqrt{m})$
static long long circle4count(const std::vector<std::pair<int, int>>&
edge, int n) {
  std::vector<int> d(n), c(n, -1), id(n);
  for (auto [u, v] : edge) ++d[u], ++d[v];
  std::iota(id.begin(), id.end(), 0);
  std::sort(id.begin(), id.end(), [&](int i, int j) {
    return d[i] < d[j] || (d[i] == d[j] && i < j);
  });
  std::vector<std::vector<int>> e(n);
  for (auto [u, v] : edge) {
    e[u].emplace_back(v);
    e[v].emplace_back(u);
  }
  // x -> y -> z and x -> w -> z
  long long ans = 0;
  for (int i = 0; i < n; ++i) {
    for (auto u : e[i]) if (id[i] < id[u]) {
      for (auto v : e[u]) if (id[i] < id[v]) ans += c[v]++;
    }
    for (auto u : e[i]) if (id[i] < id[u]) {
      for (auto v : e[u]) if (id[i] < id[v]) c[v] = 0;
    }
  }
  return ans;
}
// https://www.luogu.com.cn/blog/221955/san-yuan-huan-si-yuan-huan-ji-
shuo
```

Graph/dijkstra.cpp

```
constexpr long long INF=1e18;
template<typename G>
vector<long long> dijkstra(const G& g, int start) {
    vector dis(g.size(), INF);
    // vector<pii> pre[N];
    using node=pair<long long, int>;
    priority_queue<node, vector<node>, greater<>> q;
    dis[start] = 0;
    q.emplace(0, start);
    while (!q.empty()) {
        auto [d, u] = q.top();
        q.pop();
        if (d != dis[u]) continue;
        for (auto [v, cost] : g[u]) {
            if (dis[v] > dis[u] + cost) {
                dis[v] = dis[u] + cost;
                // pre[v].clear();
                // pre[v].pb({cost,u});
                q.emplace(dis[v], v);
            }
            // else if(dis[v]==dis[u]+cost)
            // pre[v].pb({cost,u});
        }
    }
    return dis;
}
// dijkstra for small edge weight (less than 10) aka 1-k bfs
vector<int> SmallDijkstra(const vector<vector<pair<int, int>>>& g, int
src, int lim) {
    vector<vector<int>> qs(lim);
    vector<int> dis(g.size(), -1);
    dis[src] = 0;     qs[0].push_back(src);
    for (int d = 0, maxd = 0; d <= maxd; ++d) {
        for (auto& q = qs[d % lim]; q.size(); ) {
            int u = q.back();
            q.pop_back();
            if (dis[u] != d) continue;
            for (auto [v, c] : g[u]) {
                if (dis[v] != -1 && dis[v] <= d + c) continue;
                dis[v] = d + c;
                qs[(d + c) % lim].push_back(v);
                maxd = max(maxd, d + c);
            }
        }
    }
    return dis;
}
```

Graph/dinic.cpp

```
// indexed from 0!
struct Dinic {
    static constexpr int INF = 1e9;
    int n;
    struct Edge {
        int to, cap;
        Edge(int to, int cap) : to(to), cap(cap) {}
    };
    vector<Edge> e;
    vector<std::vector<int>> g;
    vector<int> cur, h; // h = shortest distance from source, calculated
in bfs
    // after computing flow, edge (u, v) such that h[u]!=-1 and h[v]==-1
are part of min cut
    Dinic(int n) : n(n), g(n) {}
    bool bfs(int s, int t) {
        h.assign(n, -1);
        std::queue<int> que;
        h[s] = 0;
        que.push(s);
        while (!que.empty()) {
            int u = que.front();
            que.pop();
            for (int i : g[u]) {
                auto [v, c] = e[i];
```

```cpp
                if (c > 0 && h[v] == -1) {
                    h[v] = h[u] + 1;
                    if (v == t) return true;
                    que.push(v);
                }
            }
        }
        return false;
    }
    int dfs(int u, int t, int f) {
        if (u == t) return f;
        int r = f;
        for (int &i = cur[u]; i < int(g[u].size()); ++i) {
            int j = g[u][i];
            auto [v, c] = e[j];
            if (c > 0 && h[v] == h[u] + 1) {
                int a = dfs(v, t, std::min(r, c));
                e[j].cap -= a;
                e[j ^ 1].cap += a;
                r -= a;
                if (r == 0) return f;
            }
        }
        return f - r;
    }
    void addEdge(int u, int v, int c) {
        g[u].push_back((int)e.size());
        e.emplace_back(v, c);
        g[v].push_back((int)e.size());
        e.emplace_back(u, 0);
    }
    int maxFlow(int s, int t) {
        int ans = 0;
        while (bfs(s, t)) {
            cur.assign(n, 0);
            ans += dfs(s, t, INF);
        }
        return ans;
    }
};
```

Graph/dsu_on_tree.cpp

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n;
    vector<vector<int>> g(n);

    vector<int> sz(n, 1), big(n, -1);
    auto cal_size = [&](auto &slf, int u, int p) -> void {
        for (auto v : g[u]) {
            if (v == p)
                continue;
            slf(slf, v, u);
            sz[u] += sz[v];
            if (big[u] == -1 || sz[v] > sz[big[u]]) {
                big[u] = v;
            }
        }
    };
    cal_size(cal_size, 0, 0);
    auto modify = [&](auto &slf, int u, int p, int add) -> void {
        if (add) {
            // add u to result
        } else {
            // remove u from result
        }
        for (auto v : g[u]) {
            if (v == p)
                continue;
            slf(slf, v, u, add);
        }
    };
    auto dfs = [&](auto &slf, int u, int p) -> void {
        for (auto v : g[u]) {
            if (v == p || v == big[u])
                continue;
            slf(slf, v, u);          // tranverse light child
            modify(modify, v, u, 0); // remove light child
        }
        if (big[u] != -1) {
            slf(slf, big[u], u); // remove heavy child
        }
        for (auto v : g[u]) {
            if (v == p || v == big[u])
                continue;
            modify(modify, v, u, 1); // add light child again
        }
        // add u to result
        // now we have the result for subtree of u
    };
    dfs(dfs, 0, 0);
}
```

Graph/eulerian_path.cpp

```cpp
struct Eulerian {
    int n, edge_cnt = 0;
    std::vector<std::vector<std::pair<int, int>>> g;
    std::vector<int> path, deg;
    std::vector<bool> used;
    Eulerian(int _n) : n(_n), g(n), deg(n) {}
    void add_edge(int u, int v) {
        g[u].emplace_back(v, edge_cnt);
        g[v].emplace_back(u, edge_cnt);
        deg[u]++, deg[v]++;
        edge_cnt++;
    }
    void dfs(int u) {
        while (!g[u].empty()) {
            auto [v, edge] = g[u].back();
            g[u].pop_back();
            if (used[edge]) continue;
            used[edge] = true;
            dfs(v);
        }
        path.push_back(u);
    }
    std::vector<int> find_cycle(int start) {
        for (auto x : deg)
            if (x % 2) return {};
        used.resize(edge_cnt);
        dfs(start);
        if ((int)path.size() != edge_cnt + 1)
            return {}; // the graph is not connected
        reverse(path.begin(), path.end());
        return path;
    }

    std::vector<int> find_path() {
        std::vector<int> odd_deg;
        for (int i = 0; i < n; i++) {
            if (deg[i] % 2) {
                odd_deg.push_back(i);
            }
        }
        if (odd_deg.size() != 2) {
            return {};
        }
        add_edge(odd_deg[0], odd_deg[1]);
        auto res = find_cycle(odd_deg[1]);
        if (!empty(res))
            res.erase(res.begin()); // the first edge has to be the
newly added edge
        return res;
    }

    // returns:
    // - 0 if neither path nor cycle exists
    // - 1 if cycle exists
    // - 2 if path exists
    int exist() {
        int cnt = 0;
        for (int i = 0; i < n; i++) {
            if (deg[i] % 2) {
                cnt++;
            }
        }
        if (cnt == 0) {
            return 1;
        } else if (cnt == 2) {
            return 2;
        } else {
            return 0;
        }
    }
};
```

Graph/eulerian_path_directed.cpp

```
struct Eulerian_path {
    int n, edge_cnt = 0;
    std::vector<std::vector<std::pair<int, int>>> g;
    std::vector<int> path, indeg, outdeg;
    std::vector<bool> used;
    Eulerian_path(int _n) : n(_n), g(n), indeg(n), outdeg(n) {}
    void add_edge(int u, int v) {
        g[u].emplace_back(v, edge_cnt);
        indeg[v]++, outdeg[u]++;
        edge_cnt++;
    }
    void dfs(int u) {
        while (!g[u].empty()) {
            auto [v, edge] = g[u].back();
            g[u].pop_back();
            if (used[edge]) continue;
            used[edge] = true;
            dfs(v);
        }
        path.push_back(u);
    }
    std::vector<int> solve(int start) {
        for (int i = 0; i < n; i++)
            if (indeg[i] != outdeg[i]) return {};
        used.resize(edge_cnt);
        dfs(start);
        if ((int)path.size() != edge_cnt + 1)
            return {}; // the graph is not connected
        reverse(path.begin(), path.end());
        return path;
    }

    std::vector<int> solve(int start, int end) {
        add_edge(start, end);
        auto res = solve(end);
        if (!empty(res))
            res.erase(res.begin()); // the first edge has to be the
newly
                                // added edge
        return res;
    }
};
```

Graph/hamiltonian-cycle.hpp

```
#include <vector>

struct HamiltonianCycle {
    int n;
    std::vector<std::vector<int>> g;
    std::vector<int> dp, incident;

    HamiltonianCycle(int _n, std::vector<std::vector<int>> _g)
        : n(_n), g(_g), dp(1 << n), incident(n) {
        assert(g.size() == n);
        for (int i = 0; i < n; i++) {
            assert(g[i].size() == n);
        }

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                incident[i] |= g[i][j] << j;
            }
        }
        for (int msk = 1; msk < (1 << n); msk++) {
            for (int b = __builtin_ctz(msk) + 1; b < n; b++) {
                if (msk >> b & 1) {
                    dp[msk] |= bool(dp[msk ^ (1 << b)] & incident[b]) <<
b;
                }
            }
        }
    }

    bool has_cycle(int mask) {
        assert(mask >= 0 && mask < (1 << n));
        return dp[mask] & incident[__builtin_ctz(mask)];
    }

    bool has_cycle() {
        return has_cycle((1 << n) - 1);
    }

    std::vector<int> find_cycle(int mask) {
        assert(mask >= 0 && mask < (1 << n));
        int fi = __builtin_ctz(mask);
        if (!dp[mask] || ((dp[mask] & incident[fi]) == 0)) return {};

        int next = fi;
        std::vector<int> path;
        while (mask) {
            int i = __builtin_ctz(dp[mask] & incident[next]);
            path.push_back(i);
            mask ^= (1 << i);
            next = i;
        }

        return path;
    }
};
```

Graph/heavy-light_decomp.cpp

```
#include <vector>

using namespace std;

struct hld {
    int n;
    vector<int> pa, head, pos;
    int cnt = 0;

    hld(vector<vector<int>> &g, int root = 0)
      : n((int)g.size()), pa(n), head(n, -1), pos(n) {
        assert(root < (int)g.size());
        pa[root] = root;
        auto dfs = [&](auto &slf, int u) -> int {
            // we use head array as heavy child here to save some space
            int size = 1, max_size = 0;
            for (int v : g[u]) {
                if (v != pa[u]) {
                    pa[v] = u;
                    int csize = slf(slf, v);
                    size += csize;
                    if (csize > max_size) {
                        max_size = csize;
                        head[u] = v;
                    }
                }
            }
            return size;
        };
        dfs(dfs, root);

        auto dfs2 = [&](auto &slf, int u, int h) -> void {
            int hc = exchange(head[u], h);
            pos[u] = cnt++;
            if (hc == -1)
                return;
            slf(slf, hc, h);
            for (int v : g[u]) {
                if (v != pa[u] && v != hc) {
                    slf(slf, v, v);
                }
            }
        };
        dfs2(dfs2, root, root);
    }
    // decompose path from u to v into segment of [l, r] and call
process_range
    // use (r > min(pos[u], pos[v])) to test if the segment is from
right
    template <typename F>
    int decompose(int u, int v, F&& process_range, bool ignore_lca =
false) {
        while (true) {
            if (pos[u] > pos[v]) {
                swap(u, v);
            }
            if (head[u] == head[v]) break;
            int h = head[v];
            process_range(pos[h], pos[v]);
            v = pa[h];
        }
        int l = pos[u] + ignore_lca, r = pos[v];
        if (l <= r) {
            process_range(l, r);
        }
        return v;
    }
};
```

Graph/hungarian.cpp

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

// a is the adjacency matrix where a[i][j] is the cost of mathcing i-th
vertex
// in the left to the j-th vertex in the right
// It finds the minimum matching, negate the weight to find maximum
matching
// returns {cost, matching} Use a[i][matching[i]] == 0 to test if i-th
vertex
// is matched
// Time: O(n^2M)
template<class T>
pair<T, vector<int>> hungarian(const vector<vector<T>> &a) {
    if (a.empty()) return {0, {}};
    int n = a.size() + 1, m = a[0].size() + 1;
    assert(m >= n);
    vector<T> u(n), v(m); // 顶标
    vector<int> p(m), ans(n - 1);
    for (int i = 1; i < n; i++) {
        p[0] = i;
        int j0 = 0;
        vector<T> dist(m, numeric_limits<T>::max());
        vector<int> pre(m, -1);
        vector<bool> done(m + 1);
        do { // dijkstra
            done[j0] = true;
            int i0 = p[j0], j1;
            T delta = numeric_limits<T>::max();
            for (int j = 1; j < m; j++)
                if (!done[j]) {
                    auto cur = a[i0 - 1][j - 1] - u[i0] - v[j];
                    if (cur < dist[j]) dist[j] = cur, pre[j] = j0;
                    if (dist[j] < delta) delta = dist[j], j1 = j;
                }
            for (int j = 0; j < m; j++) {
                if (done[j]) u[p[j]] += delta, v[j] -= delta;
                else dist[j] -= delta;
            }
            j0 = j1;
        } while (p[j0]);
        while (j0) { // update alternating path
            int j1 = pre[j0];
            p[j0] = p[j1], j0 = j1;
        }
    }
    for (int j = 1; j < m; j++) {
        if (p[j]) ans[p[j] - 1] = j - 1;
    }
    return {-v[0], ans}; // min cost
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int l, r, m;
    cin >> l >> r >> m;
    vector g(l, vector<ll>(max(l, r), 0));

    while (m--) {
        int u, v;
        ll w;
        cin >> u >> v >> w;
        u--, v--;
        g[u][v] = min(g[u][v], -w);
    }

    auto [ans, res] = hungarian(g);
    cout << -ans << '\n';
    for (int i = 0; i < l; i++) {
        int v = res[i];
        cout << (g[i][v] == 0 ? 0 : v + 1) << " \n"[i == l - 1];
    }
    return 0;
}
```

Graph/push-relabel-mincost.hpp

```
#include <vector>
#include <limits>
#include <queue>
using namespace std;
template <typename flow_t = int, typename cost_t = int> struct mcSFlow {
    struct Edge {
        int to;
        cost_t c;
        flow_t f;
        int rev;
    };
    static constexpr cost_t INF = numeric_limits<cost_t>::max() / 2;
    static constexpr int scale = 2;
    cost_t eps = 0;
    int n, s, t;
    vector<vector<Edge>> g;
    vector<int> isq, cur;
    vector<flow_t> ex;
    vector<cost_t> h;
    vector<vector<int>> hs;
    vector<int> co;
    mcSFlow(int n, int s, int t) : n(n), s(s), t(t), g(n) {}
    void add_edge(int a, int b, cost_t cost, flow_t cap) {
        assert(cap >= 0);
        assert(a >= 0 && a < n && b >= 0 && b < n);
        if (a == b) {
            assert(cost >= 0);
            return;
        }
        cost *= n;
        eps = max(eps, abs(cost));
        g[a].emplace_back(b, cost, cap, g[b].size());
        g[b].emplace_back(a, -cost, 0, g[a].size() - 1);
    }
    void add_flow(Edge &e, flow_t f) {
        auto &back = g[e.to][e.rev];
        if (!ex[e.to] && f) hs[h[e.to]].push_back(e.to);
        e.f -= f;
        ex[e.to] += f;
        back.f += f;
        ex[back.to] -= f;
    }
    flow_t max_flow() {
        ex.assign(n, 0);
        h.assign(n, 0);
        hs.resize(2 * n);
        co.assign(2 * n, 0);
        cur.assign(n, 0);
        h[s] = n;
        ex[t] = 1;
        co[0] = n - 1;
        for (auto &e : g[s]) add_flow(e, e.f);
        if (hs[0].size()) {
            for (cost_t hi = 0; hi >= 0;) {
                int u = hs[hi].back();
                hs[hi].pop_back();
                while (ex[u] > 0) { // discharge u
                    if (cur[u] == g[u].size()) {
                        h[u] = 1e9;
                        for (int i = 0; i < g[u].size(); ++i) {
                            auto &e = g[u][i];
                            if (e.f && h[u] > h[e.to] + 1) {
                                h[u] = h[e.to] + 1, cur[u] = i;
                            }
                        }
                        if (++co[h[u]], !--co[hi] && hi < n) {
                            for (int i = 0; i < n; ++i) {
                                if (hi < h[i] && h[i] < n) {
                                    --co[h[i]];
                                    h[i] = n + 1;
                                }
                            }
                        }
                        hi = h[u];
                    } else if (g[u][cur[u]].f && h[u] == h[g[u]
[cur[u]].to] + 1) {
                        add_flow(g[u][cur[u]], min(ex[u], g[u]
[cur[u]].f));
                    } else {
                        ++cur[u];
                    }
                }
                while (hi >= 0 && hs[hi].empty()) --hi;
            }
        }
        return -ex[s];
    }
    void push(Edge &e, flow_t x) {
        if (e.f < x) x = e.f;
        e.f -= x;
        ex[e.to] += x;
        g[e.to][e.rev].f += x;
        ex[g[e.to][e.rev].to] -= x;
    }
    void relabel(int v) {
        cost_t nh = -INF; // new height
        for (int i = 0; i < g[v].size(); ++i) {
            const auto &e = g[v][i];
            if (e.f && nh < h[e.to] - e.c) {
                nh = h[e.to] - e.c;
                cur[v] = i;
            }
        }
        h[v] = nh - eps;
    }
    pair<flow_t, cost_t> minCostMaxFlow() {
        cost_t cost = 0;
        for (int i = 0; i < n; ++i)
            for (auto &e : g[i])
                cost += e.c * e.f;
        // find max-flow
        flow_t flow = max_flow();
        h.assign(n, 0);
        ex.assign(n, 0);
        isq.assign(n, 0);
        cur.assign(n, 0);
        queue<int> q;
        for (; eps; eps >>= scale) {
            // refine
            fill(cur.begin(), cur.end(), 0);
            for (int i = 0; i < n; ++i) {
                for (auto &e : g[i]) {
                    if (h[i] + e.c - h[e.to] < 0 && e.f) push(e, e.f);
                }
            }
            for (int i = 0; i < n; ++i) {
                if (ex[i] > 0) {
                    q.push(i);
                    isq[i] = 1;
                }
            }
            // make flow feasible
            while (!q.empty()) {
                int u = q.front();
                q.pop();
                isq[u] = 0;
                while (ex[u] > 0) {
                    if (cur[u] == g[u].size()) relabel(u);
                    for (int &i = cur[u], max_i = g[u].size(); i <
max_i; ++i) {
                        auto &e = g[u][i];
                        if (h[u] + e.c - h[e.to] < 0) {
                            push(e, ex[u]);
                            if (ex[e.to] > 0 && isq[e.to] == 0) {
                                q.push(e.to);
                                isq[e.to] = 1;
                            }
                            if (ex[u] == 0) break;
                        }
                    }
                }
            }
            if (eps > 1 && eps >> scale == 0) {
                eps = 1 << scale;
            }
        }
        for (int i = 0; i < n; ++i) {
            for (auto &e : g[i]) {
                cost -= e.c * e.f;
            }
        }
        return {flow, cost / 2 / n};
    }
    flow_t getFlow(Edge const &e) { return g[e.to][e.rev].f; }
};
```

Graph/push-relabel.cpp

```
/*
        Push Relabel O(n^3) implimentation using FIFO method to chose
push
   vertex. This uses gapRelabel heuristic to fasten the process even
further. If
   only the maxFlow value is required then the algo can be stopped as
soon as
   the gap relabel method is called. However, to get the actual flow
values in
   the edges, we need to let the algo terminate itself.
        This implimentation assumes zero based vertex indexing. Edges to
the
   graph can be added using the addEdge method only. capacity for
residual edges
   is set to be zero. To get the actual flow values iterate through the
edges
   and check for flow for an edge with cap > 0.
        This implimentaion is superior over dinic's for graphs where
graph is
   dense locally at some places and mostly sparse. For randomly
generated
   graphs, this implimentation gives results within seconds for n =
10000 nodes,
   m = 1000000 edges.
        Code Tested on : SPOJ FASTFLOW
        @author : triveni
*/
typedef int fType;
struct edge {
    int from, to;
    fType cap, flow;
    edge(int from, int to, fType cap, fType flow = 0)
```

```
        : from(from), to(to), cap(cap), flow(flow) {}
};
struct PushRelabel {
    int N;
    vector<edge> edges;
    vector<vector<int>> G;
    vector<int> h, inQ, count;
    vector<fType> excess;
    queue<int> Q;
    PushRelabel(int N) : N(N), count(N << 1), G(N), h(N), inQ(N),
excess(N) {}
    void addEdge(int from, int to, int cap) {
        G[from].push_back(edges.size());
        edges.push_back(edge(from, to, cap));
        G[to].push_back(edges.size());
        edges.push_back(edge(to, from, 0));
    }
    void enQueue(int u) {
        if (!inQ[u] && excess[u] > 0) Q.push(u), inQ[u] = true;
    }
    void Push(int edgeIdx) {
        edge &e = edges[edgeIdx];
        int toPush = min<fType>(e.cap - e.flow, excess[e.from]);
        if (toPush > 0 && h[e.from] > h[e.to]) {
            e.flow += toPush;
            excess[e.to] += toPush;
            excess[e.from] -= toPush;
            edges[edgeIdx ^ 1].flow -= toPush;
            enQueue(e.to);
        }
    }
    void Relabel(int u) {
        count[h[u]] -= 1;
        h[u] = 2 * N - 2;
        for (int i = 0; i < G[u].size(); ++i) {
            edge &e = edges[G[u][i]];
            if (e.cap > e.flow) h[u] = min(h[u], h[e.to]);
        }
        count[++h[u]] += 1;
    }
    void gapRelabel(int height) {
        for (int u = 0; u < N; ++u)
            if (h[u] >= height && h[u] < N) {
                count[h[u]] -= 1;
                count[h[u] = N] += 1;
                enQueue(u);
            }
    }
    void Discharge(int u) {
        for (int i = 0; excess[u] > 0 && i < G[u].size(); ++i) {
            Push(G[u][i]);
        }
        if (excess[u] > 0) {
            if (h[u] < N && count[h[u]] < 2) gapRelabel(h[u]);
            else
                Relabel(u);
        } else if (!Q.empty()) { // dequeue
            Q.pop();
            inQ[u] = false;
        }
    }
    fType getFlow(int src, int snk) {
        h[src] = N;
        inQ[src] = inQ[snk] = true;
        count[0] = N - (count[N] = 1);
        for (int i = 0; i < G[src].size(); ++i) {
            excess[src] += edges[G[src][i]].cap;
            Push(G[src][i]);
        }
        while (!Q.empty()) {
            Discharge(Q.front());
        }
        return excess[snk];
    }
};
int main() {
    int n, m;
    scanf("%d %d", &n, &m);
    PushRelabel df(n);
    while (m--) {
        int x, y, c;
        // cin >> x >> y >> c; // 0- based index
        scanf("%d%d%d", &x, &y, &c);
        --x, --y;
        if (x != y) {
            df.addEdge(x, y, c);
            df.addEdge(y, x, c);
        }
    }
    cout << df.getFlow(0, n - 1) << "\n";
    return 0;
}
```

Graph/tarjan_SCC.cpp

```
// Note that strictly speaking this is not the original tarjan's
algorithm
// because we use a slightly different definition for lowlink. However
this
// algorithm is still correctly and easier to code.
// See: https://cs.stackexchange.com/questions/96635/tarjans-scc-
example-showing-necessity-of-lowlink-definition-and-calculation-r?rq=1

#include <vector>
```

```
// Find strongly connected components of graph g. Components are
numbered in reverse topological
// order, starting from 0. It returns the number of components and an
array which indicates whichcomponent
// component each vertex belongs to
inline auto scc(const std::vector<std::vector<int>>& g) ->
std::pair<int, std::vector<int>> {
    int n = (int)size(g);
    int pos = 0;
    std::vector<bool> on_stk(n);
    std::vector<int> low(n), ord(n, -1), color(n), stk;
    int cnt = 0;

    auto dfs = [&](auto& slf, int u) -> void {
        low[u] = ord[u] = pos++;
        stk.push_back(u);
        on_stk[u] = true;
        for (auto v : g[u]) {
            if (ord[v] == -1) slf(slf, v);
            if (on_stk[v]) low[u] = std::min(low[u], low[v]);
        }
        if (low[u] == ord[u]) {
            while (true) {
                int v = stk.back();
                stk.pop_back();
                on_stk[v] = false;
                color[v] = cnt;
                if (u == v) break;
            }
            cnt++;
        }
    };

    for (int i = 0; i < n; i++) {
        if (ord[i] == -1) {
            dfs(dfs, i);
        }
    }

    return {cnt, color};
}
```

Graph/two_edge_connected_components.cpp

```
struct TECC {
    int n, pos=0;
    vector<int> ord, low, color; // order, low link, belong to which
component
    vector<vector<int>> g, comp; // graph, component
    TECC(int n) : n(n), ord(n, -1), low(n), color(n, -1), g(n) {}
    void add_edge(int u, int v) {
        g[u].emplace_back(v);
        g[v].emplace_back(u);
    }
    bool is_bridge(int u, int v) {
        if (ord[u] > ord[v]) swap(u, v);
        return ord[u] < low[v];
    }
    void dfs(int u, int p) {
        ord[u] = low[u] = pos++;
        int cnt = 0;
        for (int v : g[u]) {
            // in case there're repeated edges, only skip the first one
            if (v == p && cnt == 0) {
                cnt++;
                continue;
            }
            if (ord[v] == -1) dfs(v, u);
            low[u] = min(low[u], low[v]);
        }
    }
    void fill_component(int u) {
        comp.back().emplace_back(u);
        for (int v : g[u]) {
            if (color[v] != -1 || is_bridge(v, u)) continue;
            color[v] = color[u];
            fill_component(v);
        }
    }
    int build() {
        for (int i = 0; i < n; i++)
            if (ord[i] == -1) dfs(i, i);
        int k = 0;
        for (int i = 0; i < n; i++) {
            if (color[i] != -1) continue;
            color[i] = k++;
            comp.emplace_back();
            fill_component(i);
        }
        return k;
    }
};
int main() {
    int n, m;
    cin >> n >> m;
    TECC g(n);
    for (int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;
        g.add_edge(a, b);
    }
    int k = g.build();
    cout << k << '\n';
    for (int i = 0; i < k; i++) {
        cout << g.comp[i].size() << ' ';
        for (int v : g.comp[i])
            cout << v << ' ';
    }
    return 0;
}
```

Graph/virtual_tree.hpp

```
#pragma once
#include "graph/euler_lca.hpp"
#include <vector>

struct VirtualTree {
    int n;
    EulerLCA lca;
    std::vector<std::vector<int>> tree;

    VirtualTree(const std::vector<std::vector<int>> &g, int root)
        : n((int)g.size()), lca(g, root), tree(n) {}

    auto build_tree(const std::vector<int> &vertices)
        -> std::pair<int, const std::vector<std::vector<int>> &> {
        auto v(vertices);
        std::sort(v.begin(), v.end(), [&](int u, int v) { return
lca.pos[u] < lca.pos[v]; });
        int len = (int)v.size();
        for (int i = 1; i < len; i++) {
            v.push_back(lca.lca(v[i - 1], v[i]));
        }
        std::sort(v.begin(), v.end(), [&](int u, int v) { return
lca.pos[u] < lca.pos[v]; });
        v.erase(std::unique(v.begin(), v.end()), v.end());
        for (int i = 1; i < (int)v.size(); i++) {
            tree[lca.lca(v[i - 1], v[i])].push_back(v[i]);
        }
        return {v[0], tree};
    }

    void clear(const std::vector<int> v) {
        for (auto u : v) {
            tree[u].clear();
        }
    }

    void clear(int root) {
        for (auto v : tree[root]) {
            clear(v);
        }
        tree[root].clear();
    }
};
```

Math/BSGS.cpp

```
// solve a^x=b(mod n), 0<= x <n
#define MOD 76543
int hs[MOD], head[MOD], next[MOD], id[MOD], top;
void insert(int x, int y) {
    int k = x % MOD;
    hs[top] = x, id[top] = y, next[top] = head[k], head[k] = top++;
}
int find(int x) {
    int k = x % MOD;
    for (int i = head[k]; i != -1; i = next[i])
        if (hs[i] == x) return id[i];
    return -1;
}
int BSGS(int a, int b, int n) {
    memset(head,-1, sizeof(head));
    top = 1;
    if (b == 1) return 0;
    int m = sqrt(n * 1.0), j;
    long long x = 1, p = 1;
    for (int i = 0; i < m; ++i, p = p * a % n)
        insert(p * b % n, i);
    for (long long i = m;; i += m) {
        if ((j = find(x = x * p % n)) != -1) return i-j;
        if (i > n) break;
    }
    return -1;
}
```

Math/ChineseRT.cpp

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
#include "exGCD.hpp"

using ll = long long;
// Solve linear congruences equation:
// coef[i] * x % mod[i] = reminder[i] (mi don't need to be co-prime)
// M - lcm, x - smalleset integer solution
bool CRT(const vector<ll>& coef, const vector<ll> &rem, const vector<ll>
&mod, ll &x, ll &lcm) {
    int n = (int)coef.size();
    x = 0, lcm = 1;
    for (int i = 0; i < n; i++) {
        ll a = coef[i] * lcm, b = rem[i] - coef[i] * x, m = mod[i];
        auto [y, t, g] = exgcd(a, m);
        if (b % g) return false;
        b /= g;
        m /= g;
        x += lcm * (__int128_t(y) * b % m);
        lcm *= m;
    }
    x = (x + lcm) % lcm;
    return true;
}
```

Math/Lagrange_interpolation.hpp

```
#pragma once
// Lagrange Interpolation

#include <vector>
#include "math/combinatorics.hpp"

// Evaluate Lagrange polynomial interpolating consecutive x values at
x_eval in O(n) time
// Tested on https://codeforces.com/contest/622/problem/F
template <typename T, typename U>
static T linear_lagrange_interpolation(int x_start, const
std::vector<U>& y, int x_eval) {
    T ans{};
    const int n = (int)size(y);
    static Combi<T> c(n);

    std::vector<T> pre(n + 1), suf(n + 1);
    pre[0] = suf[n] = 1;
    for (int i = 0; i < n; i++) {
        pre[i + 1] = pre[i] * (x_eval - (x_start + i));
    }
    for (int i = n - 1; i >= 0; i--) {
        suf[i] = suf[i + 1] * (x_eval - (x_start + i));
    }

    for (int i = 0; i < n; i++) {
        auto numerator = pre[i] * suf[i + 1];
        auto denominator = T((n - i) % 2 ? 1 : -1) * c.invfac[i] *
c.invfac[n - 1 - i];
        ans += numerator * denominator * y[i];
    }

    return ans;
}

// Regular Lagrange Interpolation
// Tested on: https://www.luogu.com.cn/problem/P4781
template <typename T, typename U>
static T lagrange_interpolation(const std::vector<U>& x, const
std::vector<U>& y, int x_eval) {
    T ans{};
    const int n = (int)size(x);

    for (int i = 0; i < n; i++) {
        T numerator = y[i];
        T denominator = 1;
        for (int j = 0; j < n; j++) {
            if (j == i) continue;
            numerator *= (x_eval - x[j]);
            denominator *= (x[i] - x[j]);
        }
        ans += numerator / denominator;
    }

    return ans;
}
```

Math/binomial.cpp

```
#include <vector>
using namespace std;
inline namespace binomial {
    using T = mint;
    // using T = long long;
    vector<vector<T>> binom;
    void init(int n) {
        binom.resize(n+1, vector<T>(n+1));
        binom[0][0]=1;
        for (int i=1; i<=n; i++) {
            binom[i][0]=binom[i][i]=1;
            for (int j=1; j<i; j++)
                binom[i][j]=binom[i-1][j]+binom[i-1][j-1];
        }
    }

    T C(int n, int m) { // n choose m
        if (m<0 || m>n) return T{};
        return binom[n][m];
    }
}
```

Math/euler.h

```
#define NEGPOW(e) ((e) % 2 ? -1 : 1)

int jacobi(int a, int m) {
    if (a == 0) return m == 1 ? 1 : 0;
    if (a % 2)  return NEGPOW((a-1)*(m-1)/4)*jacobi(m%a, a);
    else return NEGPOW((m*m-1)/8)*jacobi(a/2, m);
}

int invMod(int a, int m) {
    int x, y;
    if (extgcd(a, m, x, y) == 1) return (x + m) % m;
    else return 0; // unsolvable
}

// No solution when: n(p-1)/2 = -1 mod p
```

```
int sqrtMod(int n, int p) {
  int S, Q, W, i, m = invMod(n, p);
  for (Q = p - 1, S = 0; Q % 2 == 0; Q /= 2, ++S);
  do { W = rand() % p; } while (W == 0 || jacobi(W, p) != -1);
  for (int R = powMod(n, (Q+1)/2, p), V = powMod(W, Q, p); ;) {
    int z = R * R * m % p;
    for (i = 0; i < S && z % p != 1; z *= z, ++i);
    if (i == 0) return R;
    R = (R * powMod(V, 1 << (S-i-1), p)) % p;
  }
}

bool eulercriterion(int n, int p) {
  if(powMod(n, (p-1)/2, p) == 1) return true;
  return false;
}

int powMod(int a, int b, int p) {
  int res=1;
  while(b) {
    if(b&1) res=int( res * 1ll * a % p), --b;
    else a=int (a * 1ll * a%p), b>>=1;    }
  return res;
}
```

Math/exGCD.hpp

```
#include<bits/stdc++.h>
using ll = long long;
using namespace std;
// Returns {x, y, g} which is a solution to a * x + b * y = g = gcd(a,
b)
static array<ll, 3> exgcd(ll a, ll b) {
    if (b == 0) return {1, 0, a};
    auto [x, y, g] = exgcd(b, a % b);
    return {y, x - a / b * y, g};
}

/*
 * Solves a * x + b * y = c, equivalently a * x = c (mod b)
 * returns {x, y, g} where x is the smallest non-negative solution
 * and g is gcd(a, b), or returns {} if the solution doesn't exist
 * all solutions: x = x0 + k * b / g, y = y0 - k * a / g
 * smallest non-negative x = (x0 % t + t) % t, where t = b / g
 */
static array<ll, 3> liEu(ll a, ll b, ll c) {
    auto [x, y, g] = exgcd(a, b);
    if (c % g != 0) return {};
    // smallest positive x:
    int64_t t = b / g;
    x = (x * ((c / g) % t) % t + t) % t;
    y = (c - a * x) / b;
    return std::array{x, y, g};
}
```

Math/factorization.cpp

```
namespace Fractorization {
    using u64 = uint64_t;
    using u128 = __uint128_t;
    using ull = unsigned long long;
    mt19937
rand(chrono::steady_clock::now().time_since_epoch().count());

    u64 binPow(u64 a, u64 b, u64 mod) {
        if (b == 0)
            return 1;
        if (b & 1)
            return (u128)a * binPow(a, b ^ 1, mod) % mod;
        return binPow((u128)a * a % mod, b >> 1, mod);
    }
    bool checkComp(u64 n, u64 a, u64 d, int s) {
        u64 x = binPow(a, d, n);
        if (x == 1 || x == n - 1)
            return false;
        for (int r = 1; r < s; r++) {
            x = (u128)x * x % n;
            if (x == n - 1)
                return false;
        }
        return true;
    }
    bool RabinMiller(u64 n) {
        if (n < 2)
            return false;
        int r = 0;
        u64 d = n - 1;
        while (!(d & 1))
            d >>= 1, r++;
        for (int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {
            if (n == a)
                return true;
            if (checkComp(n, a, d, r))
                return false;
        }
        return true;
    }
    ull mult(ull a, ull b, ull mod) { return (u128)a * b % mod; }
    ull rho(ull n) { // wiull find a factor < n, but not necessarily
prime
        if (~n & 1)
            return 2;
        ull c = rand() % n, x = rand() % n, y = x, d = 1;
        while (d == 1) {
            x = (mult(x, x, n) + c) % n;
            y = (mult(y, y, n) + c) % n;
            y = (mult(y, y, n) + c) % n;
            d = gcd(max(x, y) - min(x, y), n);
        }
        return d == n ? rho(n) : d;
    }
    vector<pair<ull, int>> factorRho(ull n) {
        map<ull, int> fact;
        function<void(ull)> factRho = [&](ull n) {
            if (n == 1)
                return;
            if (RabinMiller(n)) {
                fact[n]++;
                return;
            }
            ull factor = rho(n);
            factRho(factor);
            factRho(n / factor);
        };
        factRho(n);
        vector<pair<ull, int>> facts;
        for (auto &p : fact)
            facts.push_back(p);
        return facts;
    }
    vector<pair<int, int>> factor(int n) {
        vector<pair<int, int>> facts;
        for (int f = 2; f * f <= n; f++) {
            if (n % f == 0) {
                int c = 0;
                while (n % f == 0) {
                    n /= f;
                    c++;
                }
                facts.emplace_back(f, c);
            }
        }
        return facts;
    }
} // namespace Fractorization
using namespace Fractorization;
```

Math/fft.hpp

```
#include <complex>
#include <vector>

using cd = std::complex<double>;
constexpr double PI = M_PI; // from <math.h>

static void fft(std::vector<cd> &a, bool invert) {
    int n = a.size();
    // permute the array to do in-place calculation
    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j & bit; bit >>= 1)
            j ^= bit;
        j ^= bit;
        if (i < j) swap(a[i], a[j]);
    }
    for (int len = 2; len <= n; len <<= 1) {
        double ang = 2 * PI / len * (invert ? -1 : 1);
        cd wlen(cos(ang), sin(ang));
        for (int i = 0; i < n; i += len) {
            cd w(1);
            for (int j = 0; j < len / 2; j++) {
                cd u = a[i + j], v = a[i + j + len / 2] * w;
                a[i + j] = u + v;
                a[i + j + len / 2] = u - v;
                w *= wlen;
            }
        }
    }
    if (invert) {
        for (auto &x : a)
            x /= n;
    }
}

// calculates the convolution of a and b
static
std::vector<int> convolve_fft(const std::vector<int> &a, const
std::vector<int> &b) {
    std::vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());
    int n = 1 << (__lg(size(a) + size(b) - 1) + 1);
    fa.resize(n);
    fb.resize(n);

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

    std::vector<int> result(n);
    for (int i = 0; i < n; i++)
        result[i] = round(fa[i].real());
    return result;
}
```

Math/gauss.h

```
const double EPS = 1e-9;
const int INF = 2;

int gauss (vector < vector<double> > a, vector<double> & ans) {
  int n = (int) a.size();
  int m = (int) a[0].size() - 1;

  vector<int> where (m, -1);
  for (int col=0, row=0; col<m && row<n; ++col) {
    int sel = row;
    for (int i=row; i<n; ++i)
      if (abs (a[i][col]) > abs (a[sel][col]))
        sel = i;
    if (abs (a[sel][col]) < EPS)
      continue;
    for (int i=col; i<=m; ++i)
      swap (a[sel][i], a[row][i]);
    where[col] = row;

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

  ans.assign (m, 0);
  for (int i=0; i<m; ++i)
    if (where[i] != -1)
      ans[i] = a[where[i]][m] / a[where[i]][i];
  for (int i=0; i<n; ++i) {
    double sum = 0;
    for (int j=0; j<m; ++j)
      sum += ans[j] * a[i][j];
    if (abs (sum - a[i][m]) > EPS)
      return 0;
  }

  for (int i=0; i<m; ++i)
    if (where[i] == -1)
      return INF;
  return 1;
}
```

Math/inverse.h

```
const ll MOD = 998244353;
vector<ll> inv(n+1);
inv[1]=1;
for(int i = 2; i < n + 1; ++i) inv[i] = MOD - (MOD/i) * inv[MOD % i] %
MOD;
```

Math/lucas.h

```
// when n and m are big but p is small
ll Lucas(ll n, ll m, ll p) {
  if (m == 0) return 1;
  return (C(n % p, m % p, p) * Lucas(n / p, m / p, p)) % p;
}
```

Math/nfft.hpp

```
#pragma once
#include <vector>
#include "misc/util.hpp"

using ll = int64_t;
constexpr int P = 998244353;

static int power(int a, int b) {
    int res = 1;
    for (; b; b >>= 1, a = (ll)a * a % P)
        if (b & 1)
            res = (ll)res * a % P;
    return res;
}

static void dft(std::vector<int> &a) {
    static std::vector<int> rev, roots{0, 1};
    int n = a.size();
    if (int(rev.size()) != n) {
        int k = __builtin_ctz(n) - 1;
        rev.resize(n);
        for (int i = 0; i < n; ++i)
            rev[i] = rev[i >> 1] >> 1 | (i & 1) << k;
    }
    for (int i = 0; i < n; ++i)
        if (rev[i] < i)
            std::swap(a[i], a[rev[i]]);
    if (int(roots.size()) < n) {
        int k = __builtin_ctz(roots.size());
        roots.resize(n);
        while ((1 << k) < n) {
            int e = power(3, (P - 1) >> (k + 1));
            for (int i = 1 << (k - 1); i < (1 << k); ++i) {
                roots[2 * i] = roots[i];
                roots[2 * i + 1] = (ll)roots[i] * e % P;
            }
            ++k;
        }
    }
    for (int k = 1; k < n; k *= 2) {
        for (int i = 0; i < n; i += 2 * k) {
            for (int j = 0; j < k; ++j) {
                int u = a[i + j];
                int v = (ll)a[i + j + k] * roots[k + j] % P;
                int x = u + v;
                if (x >= P)
                    x -= P;
                a[i + j] = x;
                x = u - v;
                if (x < 0)
                    x += P;
                a[i + j + k] = x;
            }
        }
    }
}

static void idft(std::vector<int> &a) {
    int n = a.size();
    std::reverse(a.begin() + 1, a.end());
    dft(a);
    int inv = power(n, P - 2);
    for (int i = 0; i < n; ++i)
        a[i] = (ll)a[i] * inv % P;
}

// calculates the convolution of a and b
static
std::vector<int> convolve(const std::vector<int> &a, const
std::vector<int> &b) {
    auto fa{a}, fb{b};
    int n = 1 << (lg(size(a) + size(b) - 1) + 1);
    fa.resize(n);
    fb.resize(n);

    dft(fa);
    dft(fb);
    for (int i = 0; i < n; i++)
        fa[i] = (ll)fa[i] * fb[i] % P;
    idft(fa);

    return fa;
}
```

Math/sieve.cpp

```
struct Prime {
    std::vector<int> primes, mn_factor;
    Prime(int N) {
        mn_factor.resize(N + 1);
        for (int i = 2; i <= N; ++i) {
            if (mn_factor[i] == 0) {
                primes.push_back(i);
                mn_factor[i] = i;
            }
            for (auto p : primes) {
                if ((long long)i * p > N) break;
                mn_factor[i * p] = p;
                if (i % p == 0) break;
            }
        }
    }

    bool is_prime(int n) {
        return mn_factor[n] == n;
    }

    // Factors n in O(log(n)) time
    std::vector<std::pair<int, int>> factor(int n) {
        std::vector<std::pair<int, int>> factors;
        while (n > 1) {
            int fac = mn_factor[n], cnt = 0;
            while (n % fac == 0) {
                cnt++;
                n /= fac;
            }
            factors.emplace_back(fac, cnt);
        }
        return factors;
    };
};
```

Math/simplex.h

```
/**
 * Author: Stanford
 * Source: Stanford Notebook
 * License: MIT
 * Description: Solves a general linear maximization problem: maximize
$c^T x$ subject to $Ax \le b$, $x \ge 0$.
 * Returns -inf if there is no solution, inf if there are arbitrarily
good solutions, or the maximum value of $c^T x$ otherwise.
 * The input vector is set to an optimal $x$ (or in the unbounded case,
an arbitrary solution fulfilling the constraints).
 * Numerical stability is not guaranteed. For better performance, define
variables such that $x = 0$ is viable.
 * Usage:
 * vvd A = {{1,-1}, {-1,1}, {-1,-2}};
 * vd b = {1,1,-4}, c = {-1,-1}, x;
 * T val = LPSolver(A, b, c).solve(x);
 * Time: O(NM * \#pivots), where a pivot may be e.g. an edge relaxation.
O(2^n) in the general case.
 * Status: seems to work?
 */
```

```
typedef double T; // long double, Rational, double + mod<P>...
typedef vector<T> vd;
typedef vector<vd> vvd;

const T eps = 1e-8, inf = 1/.0;
#define ltj(X) if(s == -1 || MP(X[j],N[j]) < MP(X[s],N[s])) s=j

struct LPSolver {
  int m, n;
  vi N, B;
  vvd D;

  LPSolver(const vvd& A, const vd& b, const vd& c) :
    m(sz(b)), n(sz(c)), N(n+1), B(m), D(m+2, vd(n+2)) {
      FOR(i,0,m) FOR(j,0,n) D[i][j] = A[i][j];
      FOR(i,0,m) { B[i] = n+i; D[i][n] = -1; D[i][n+1] = b[i];}
      FOR(j,0,n) { N[j] = j; D[m][j] = -c[j]; }
      N[n] = -1; D[m+1][n] = 1;
    }

  void pivot(int r, int s) {
    T *a = D[r].data(), inv = 1 / a[s];
    FOR(i,0,m+2) if (i != r && abs(D[i][s]) > eps) {
      T *b = D[i].data(), inv2 = b[s] * inv;
      FOR(j,0,n+2) b[j] -= a[j] * inv2;
      b[s] = a[s] * inv2;
    }
    FOR(j,0,n+2) if (j != s) D[r][j] *= inv;
    FOR(i,0,m+2) if (i != r) D[i][s] *= -inv;
    D[r][s] = inv;
    swap(B[r], N[s]);
  }

  bool simplex(int phase) {
    int x = m + phase - 1;
    for (;;) {
      int s = -1;
      FOR(j,0,n+1) if (N[j] != -phase) ltj(D[x]);
      if (D[x][s] >= -eps) return true;
      int r = -1;
      FOR(i,0,m) {
        if (D[i][s] <= eps) continue;
        if (r == -1 || MP(D[i][n+1] / D[i][s], B[i])
                     < MP(D[r][n+1] / D[r][s], B[r])) r = i;
      }
      if (r == -1) return false;
      pivot(r, s);
    }
  }

  T solve(vd &x) {
    int r = 0;
    FOR(i,1,m) if (D[i][n+1] < D[r][n+1]) r = i;
    if (D[r][n+1] < -eps) {
      pivot(r, n);
      if (!simplex(2) || D[m+1][n+1] < -eps) return -inf;
      FOR(i,0,m) if (B[i] == -1) {
        int s = 0;
        FOR(j,1,n+1) ltj(D[i]);
        pivot(i, s);
      }
    }
    bool ok = simplex(1); x = vd(n);
    FOR(i,0,m) if (B[i] < n) x[B[i]] = D[i][n+1];
    return ok ? D[m][n+1] : inf;
  }
};
```

Misc/Mo's_algorithm.cpp

```
// Mo's algorithm, solve m offline queries on array of length n in O(n
sqrt(m))
struct MO {
    int n, m=0;
    struct node {
        int l, r, id;
    };
    vector<node> query;
    MO(int _n) : n(_n) {}
    void add_query(int l, int r) {
        query.push_back({l, r, m++});
    }
    template<typename F>
    vector<int> solve(F&& move) {
        const int BLOCK_SIZE = (n<=m ? ceil(sqrt(n)) : n/ceil(sqrt(m)));
        sort(query.begin(), query.end(), [&](const node& lhs, const
node& rhs) {
            if (lhs.l / BLOCK_SIZE != rhs.l / BLOCK_SIZE) return lhs.l <
rhs.l;
            return ((lhs.l / BLOCK_SIZE) & 1) ? lhs.r < rhs.r : lhs.r >
rhs.r;
        });
        vector<int> ans(m);
        int l=0, r=-1, cur=0;
        for (const auto& [ql, qr, id] : query) {
            while (l > ql) move(--l, 1, cur);
            while (r < qr) move(++r, 1, cur);
            while (l < ql) move(l++, -1, cur);
            while (r > qr) move(r--, -1, cur);
            ans[id]=cur;
        }
        return ans;
    }
};
```

```
// example: find the most occurrence in ranges
int main() {
    int n, q;
    MO mo(n);
    vector<int> a(n), counter(n+1), freq(3e5+1);
    auto ans=mo.solve([&](int i, int dir, int& cur) {
        int val=a[i];
        int c=freq[val];
        counter[c]--;
        if (dir==1) {
            freq[val]++;
            counter[freq[val]]++;
            cur=max(cur, freq[val]);
        } else {
            freq[val]--;
            counter[freq[val]]++;
            if (counter[cur]==0) cur--;
        }
    });
}
```

string/ac-automaton.cpp

```
#include <vector>
#include <array>
#include <string>
#include <queue>
/** Modified from:
 * https://github.com/kth-competitive-programming/kactl/blob/master/
content/strings/AhoCorasick.h
 * If there's no duplicated patterns, just call the constructor,
otherwise handle it beforehand
 * by yourself, or use the return value of insert
 * empty patterns are not allowed.
 * Time: construction takes $O(26N)$,
 * where $N =$ sum of length of patterns. find(x) is $O(N)$, where N =
length of
 * x. findAll is $O(N+M)$ where M is number of occurrence of all pattern
(up to N*sqrt(N)) */

template<int alpha = 26, int first = 'a'>
struct AhoCorasick {
    struct Node {
        // back: failure link, points to longest suffix that is in the
trie.
        // end: longest pattern that ends here, is -1 if no patten ends
here.
        // nmatches: number of patterns that is a suffix of current node
        // output: output link, points to the longest pattern that is a
suffix
        // of current node
        int back = 0, end = -1, nmatches = 0, output = -1;
        std::array<int, alpha> next;
        Node() { std::fill(next.begin(), next.end(), -1); }
    };

    std::vector<Node> N;

    AhoCorasick() : N(1) {}
    AhoCorasick(const std::vector<std::string>& patterns) {
        for (int i = 0; i < (int)patterns.size(); i++) {
            insert(patterns[i], i);
        }
        build();
    }

    // returns -1 if there's no duplicated pattern already in the trie
    // returns the id of the duplicated pattern otherwise
    int insert(const std::string &s, int j) { // j: id of string s
        assert(!s.empty());
        int n = 0;
        for (char c : s) {
            if (N[n].next[c - first] == -1) {
                N[n].next[c - first] = (int)N.size();
                N.emplace_back();
            }
            n = N[n].next[c - first];
        }
        if (N[n].end != -1) {
            return N[n].end;
        }
        N[n].end = j;
        N[n].nmatches++;
        return -1;
    }

    void build() {
        std::queue<int> q;
        q.push(0);
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (int i = 0; i < alpha; i++) {
                int fail = u ? N[N[u].back].next[i] : 0;
                auto v = N[u].next[i];
                if (v == -1) N[u].next[i] = fail;
                else {
                    N[v].back = fail;
                    // if prev is an end node, then set output to prev
node,
                    // otherwise set to output link of prev node
                    N[v].output = N[fail].end == -1 ? N[fail].output :
fail;
                    N[v].nmatches += N[fail].nmatches;
                    q.push(v);
                }
            }
        }
    }

    // for each position, finds the longest pattern that ends here
    std::vector<int> find(const std::string &text) {
        int len = (int)text.size();
        std::vector<int> res(len);
        int n = 0;
        for (int i = 0; i < len; i++) {
            n = N[n].next[text[i] - first];
            res[i] = N[n].end;
        }
        return res;
    }

    // for each position, finds all patterns that ends here
    std::vector<std::vector<int>> find_all(const std::string &text) {
        int len = (int)text.size();
        std::vector<std::vector<int>> res(len);
        int n = 0;
        for (int i = 0; i < len; i++) {
            n = N[n].next[text[i] - first];
            if (N[n].end != -1) {
                res[i].push_back(N[n].end);
            }
            for (int ind = N[n].output; ind != -1; ind = N[ind].output)
{
                assert(N[ind].end != -1);
                res[i].push_back(N[ind].end);
            }
        }
        return res;
    }

    // finds the number of occurrence of each pattern
    std::vector<int> find_cnt(const std::string& text, int
num_of_patterns) {
        std::vector<int> cnt(num_of_patterns);
        int p = 0;
        for (auto c : text) {
            p = N[p].next[c - first];
            if (N[p].end != -1) {
                cnt[N[p].end] += 1;
            }
            for (int i = N[p].output; i != -1; i = N[i].output) {
                cnt[N[i].end]++;
            }
        }
        return cnt;
    }
};
```

string/kmp.cpp

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

string/manacher.cpp

```
#include <array>
#include <vector>

// return [even, odd] where:
//
// even[i] is the half of the length of longest palindrome starting from
the
// i-th gap, the first gap is before the first character, there are n+1
gaps.
//
// odd[i] is half of the length of longest palindrome starting from the
i-th
// character.
template <typename T> std::array<std::vector<int>, 2> manacher(const T
&s) {
    int n = (int)size(s);
    std::array d{std::vector<int>(n + 1), std::vector<int>(n)};

    for (int z : {0, 1}) {
        auto &p = d[z];
        for (int i = 0, l = 0, r = 0; i < n; i++) {
            int t = r - i + !z;
            if (i < r) {
                p[i] = std::min(t, p[l + t]);
            }
            int l2 = i - p[i], r2 = i + p[i] - !z;
            while (l2 && r2 + 1 < n && s[l2 - 1] == s[r2 + 1]) {
                ++p[i];
                --l2, ++r2;
            }
            if (r2 > r) {
                l = l2, r = r2;
            }
        }
    }
    return d;
}
```

string/polyhash.cpp

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

using ll = long long;
using i128 = __int128;
int main() {
    const int N = 1e6;
    vector<ll> pow(N + 1);
    ll base = 233, mod = 1'000'000'000'000'000'003;
    pow[0] = 1;
    for (int i = 1; i <= N; i++) {
        pow[i] = (i128)pow[i - 1] * base % mod;
    }

    auto hash = [&](const string& s) {
        int sz = (int)size(s);
        vector<ll> pref(sz + 1);
        for (int i = 0; i < sz; i++) {
            pref[i + 1] = ((i128)pref[i] * base % mod + s[i]) % mod;
        }
        return pref;
    };

    // [l, r)
    auto substr = [&](const vector<ll>& h, int l, int r) {
        return (h[r] - (i128)h[l] * pow[r - l] % mod + mod) % mod;
    };

    auto concat = [&](ll lhs, ll rhs, int len_rhs) {
        return ((i128)lhs * pow[len_rhs] % mod + rhs) % mod;
    };

}
```

string/suffix automaton.cpp

```
// source: https://cp-algorithms.com/string/suffix-automaton.html
struct SAM {
    struct state {
        int len = 0, link = -1;
        unordered_map<char, int> next;
    };
    int last = 0; // the index of the equivalence class of the whole
string
    vector<state> st;
    void extend(char c) {
        int cur = (int)st.size();
        st.emplace_back();
        st[cur].len = st[last].len + 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 = (int)st.size();
                st.push_back(st[q]);
                st[clone].len = st[p].len + 1;
                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;
    }
    SAM() { st.emplace_back(); }
    SAM(const string &s) : SAM() {
        for (auto c : s)
            extend(c);
    }
};
```

string/suffix_array.cpp

```
#include <string>
#include <vector>
// O(n log(n)),actually calculates cyclic shifts
static std::vector<int> suffix_array(std::string s) {
    s += "#";
    int n = (int)s.size(), N = n + 256;
    std::vector<int> sa(n), ra(n);
    for (int i = 0; i < n; i++)
        sa[i] = i, ra[i] = s[i];
    for (int k = 0; k < n; k ? k *= 2 : k++) {
        std::vector<int> nsa(sa), nra(n), cnt(N);
        for (int i = 0; i < n; i++) nsa[i] = (nsa[i] - k + n) % n;
        for (int i = 0; i < n; i++) cnt[ra[i]]++;
        for (int i = 1; i < N; i++) cnt[i] += cnt[i - 1];
        for (int i = n - 1; i >= 0; i--) sa[--cnt[ra[nsa[i]]]] = nsa[i];

        int r = 0;
        for (int i = 1; i < n; i++) {
            if (ra[sa[i]] != ra[sa[i - 1]]) r++;
            else if (ra[(sa[i] + k) % n] != ra[(sa[i - 1] + k) % n]) r+
+;
            nra[sa[i]] = r;
        }
        ra = nra;
    }
    sa.erase(sa.begin());
    return sa;
}

static std::vector<int>
build_lcp(const std::string &s, const std::vector<int> &sa) { // lcp of
sa[i] and sa[i-1]
    int n = (int)s.size();
    std::vector<int> pos(n);
    for (int i = 0; i < n; i++) pos[sa[i]] = i;
    std::vector<int> lcp(n);
    for (int i = 0, k = 0; i < n; i++) {
        if (pos[i] == 0) continue;
        if (k) k--;
        while (s[i + k] == s[sa[pos[i] - 1] + k])
            k++;
        lcp[pos[i]] = k;
    }
    return lcp;
}
```

string/suffix_array_linear.cpp

```
//O(n)
vector<int> suffix_array(const string& s, int char_bound) {
    int n=s.size();
    vector<int> a(n);
    if (n == 0) return a;
    if (char_bound != -1) {
        vector<int> aux(char_bound, 0);
        for (int i = 0; i < n; i++) aux[s[i]]++;
        int sum = 0;
        for (int i = 0; i < char_bound; i++) {
            int add = aux[i];
            aux[i] = sum;
            sum += add;
        }
        for (int i = 0; i < n; i++) {
            a[aux[s[i]]++] = i;
        }
    } else {
        iota(a.begin(), a.end(), 0);
        sort(a.begin(), a.end(), [&s](int i, int j) { return s[i] <
s[j]; });
    }
    vector<int> sorted_by_second(n);
    vector<int> ptr_group(n);
    vector<int> new_group(n);
    vector<int> group(n);
    group[a[0]] = 0;
    for (int i = 1; i < n; i++) {
        group[a[i]] = group[a[i - 1]] + (!(s[a[i]] == s[a[i - 1]]));
    }
    int cnt = group[a[n - 1]] + 1;
    int step = 1;
    while (cnt < n) {
        int at = 0;
        for (int i = n - step; i < n; i++) {
            sorted_by_second[at++] = i;
        }
        for (int i = 0; i < n; i++) {
            if (a[i] - step >= 0) {
                sorted_by_second[at++] = a[i] - step;
            }
        }
        for (int i = n - 1; i >= 0; i--) {
            ptr_group[group[a[i]]] = i;
        }
        for (int i = 0; i < n; i++) {
            int x = sorted_by_second[i];
            a[ptr_group[group[x]]++] = x;
        }
        new_group[a[0]] = 0;
        for (int i = 1; i < n; i++) {
            if (group[a[i]] != group[a[i - 1]]) {
                new_group[a[i]] = new_group[a[i - 1]] + 1;
            } else {
                int pre = (a[i - 1] + step >= n ? -1 : group[a[i - 1] +
step]);
                int cur = (a[i] + step >= n ? -1 : group[a[i] + step]);
                new_group[a[i]] = new_group[a[i - 1]] + (pre != cur);
            }
        }
        swap(group, new_group);
        cnt = group[a[n - 1]] + 1;
        step <<= 1;
    }
    return a;
}
```

string/trie.cpp

```
#include <bits/stdc++.h>
using namespace std;
template <typename T> struct Trie {
    struct node {
        map<T, int> ch;
        bool is_leaf;
    };
    vector<node> t;
    Trie() { new_node(); }
    int new_node() {
        t.emplace_back();
        return (int)t.size() - 1;
    }
    template <typename S> void insert(const S &s) {
        int p = 0;
        for (int i = 0; i < (int)s.size(); i++) {
            auto ch = s[i];
            if (!t[p].ch.count(ch)) {
                t[p].ch[ch] = new_node();
            }
            p = t[p].ch[ch];
        }
        t[p].is_leaf = true;
    }
    template <typename S> bool find(const S &s) {
        int p = 0;
        for (auto ch : s) {
            if (!t[p].ch.count(ch))
                return false;
            p = t[p].ch[ch];
        }
        return t[p].is_leaf;
    }
};
```

string/z-function.cpp

```
// In other words, z[i] is the length of the longest common prefix
between s and the suffix of s starting at i.
vector<int> z_function(const string& s) {
    int n = (int)s.size();
    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;
}
```

vimrc

```
set ttymouse=sgr

syntax on
set mouse=a si noswf cin et sw=4 ts=4 sr sts=-1 nu
```