Contents

1 Basic

1.1 vimrc

```
set nu rnu cin ts=4 sw=4 autoread hls
sy on
map<leader>b :w<bar>!g++ -std=c++17 '%' -DKEV -fsanitize=undefined -o /tmp/.run<CR>
map<leader>r :w<bar>!cat 01.in && echo "---" && /tmp/.run < 01.in<CR>
map<leader>i :!/tmp/.run<CR>
map<leader>c I//<Esc>
map<leader>y :%y+<CR>
map<leader>l :%d<bar>0r ~/t.cpp<CR>
```

1.2 Default code

```cpp
#include <bits/stdc++.h>
using namespace std;
using i64 = long long;
using ll = long long;
#define SZ(v) (ll)((v).size())
#define pb emplace_back
#define AI(i) begin(i), end(i)
#define X first
#define Y second
template<class T> bool chmin(T &a, T b) { return b < a && (a = b, true); }
template<class T> bool chmax(T &a, T b) { return a < b && (a = b, true); }
#ifdef KEV
#define DE(args...) kout("[ " + string(#args) + " ] = ", args)
void kout() { cerr << endl; }
template<class T, class ...U> void kout(T a, U ...b) { cerr << a << ' ', kout(b...); }
template<class T> void debug(T l, T r) { while (l != r) cerr << *l << " \n"[next(l)==r], ++l; }
#else
#define DE(...) 0
#define debug(...) 0
#endif
int main() {
    cin.tie(nullptr)->sync_with_stdio(false);
    return 0;
}
```

1.3 Fast Integer Input

```cpp
char buf[1 << 16], *p1 = buf, *p2 = buf;
char get() {
    if (p1 == p2) {
        p1 = buf;
        p2 = p1 + fread(buf, 1, sizeof(buf), stdin);
    }
    if (p1 == p2)
        return -1;
    return *p1++;
}
char readChar() {
    char c = get();
    while (isspace(c))
        c = get();
    return c;
}
int readInt() {
    int x = 0;
    char c = get();
    while (!isdigit(c))
        c = get();
    while (isdigit(c)) {
        x = 10 * x + c - '0';
        c = get();
    }
    return x;
}
```

1.4 Pragma optimization

```
#pragma GCC optimize("Ofast", "no-stack-protector", "no-math-
    errno", "unroll-loops")
#pragma GCC target("sse,sse2,sse3,ssse3,sse4,sse4.2,popcnt,abm,
    mmx,avx,tune=native,arch=core-avx2,tune=core-avx2")
#pragma GCC ivdep
```

2 Flows, Matching

2.1 Flow

```
template <typename F>
struct Flow {
    static constexpr F INF = numeric_limits<F>::max() / 2;
    struct Edge {
        int to;
        F cap;
        Edge(int to, F cap) : to(to), cap(cap) {}
    };
    int n;
    vector<Edge> e;
    vector<vector<int>> adj;
    vector<int> cur, h;
    Flow(int n) : n(n), adj(n) {}
    bool bfs(int s, int t) {
        h.assign(n, -1);
        queue<int> q;
        h[s] = 0;
        q.push(s);
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (int i : adj[u]) {
                auto [v, c] = e[i];
                if (c > 0 && h[v] == -1) {
                    h[v] = h[u] + 1;
                    if (v == t) { return true; }
                    q.push(v);
                }
            }
        }
        return false;
    }
    F dfs(int u, int t, F f) {
        if (u == t) { return f; }
        F r = f;
        for (int &i = cur[u]; i < int(adj[u].size()); i++) {
            int j = adj[u][i];
            auto [v, c] = e[j];
            if (c > 0 && h[v] == h[u] + 1) {
                F a = dfs(v, t, min(r, c));
                e[j].cap -= a;
                e[j ^ 1].cap += a;
                r -= a;
                if (r == 0) { return f; }
            }
        }
        return f - r;
    }
    // can be bidirectional
    void addEdge(int u, int v, F cf = INF, F cb = 0) {
        adj[u].push_back(e.size()), e.emplace_back(v, cf);
        adj[v].push_back(e.size()), e.emplace_back(u, cb);
    }
    F maxFlow(int s, int t) {
        F ans = 0;
        while (bfs(s, t)) {
            cur.assign(n, 0);
            ans += dfs(s, t, INF);
        }
        return ans;
    }
    // do max flow first
    vector<int> minCut() {
        vector<int> res(n);
        for (int i = 0; i < n; i++) { res[i] = h[i] != -1; }
        return res;
    }
};
```

2.2 MCMF

```
template <class Flow, class Cost>
struct MinCostMaxFlow {
public:
    static constexpr Flow flowINF = numeric_limits<Flow>::max()
        ;
    static constexpr Cost costINF = numeric_limits<Cost>::max()
        ;
    MinCostMaxFlow() {}
    MinCostMaxFlow(int n) : n(n), g(n) {}
    int addEdge(int u, int v, Flow cap, Cost cost) {
        int m = int(pos.size());
        pos.push_back({u, int(g[u].size())});
        g[u].push_back({v, int(g[v].size()), cap, cost});
        g[v].push_back({u, int(g[u].size()) - 1, 0, -cost});
        return m;
    }
    struct edge {
        int u, v;
        Flow cap, flow;
        Cost cost;
    };
    edge getEdge(int i) {
        int m = int(pos.size());
        auto _e = g[pos[i].first][pos[i].second];
        auto _re = g[_e.v][_e.rev];
        return {pos[i].first, _e.v, _e.cap + _re.cap, _re.cap,
            _e.cost};
    }
    vector<edge> edges() {
        int m = int(pos.size());
        vector<edge> result(m);
        for (int i = 0; i < m; i++) { result[i] = getEdge(i); }
        return result;
    }
    pair<Flow, Cost> maxFlow(int s, int t, Flow flow_limit =
        flowINF) { return slope(s, t, flow_limit).back(); }
    vector<pair<Flow, Cost>> slope(int s, int t, Flow
        flow_limit = flowINF) {
        vector<Cost> dual(n, 0), dis(n);
        vector<int> pv(n), pe(n), vis(n);
        auto dualRef = [&]() {
            fill(dis.begin(), dis.end(), costINF);
            fill(pv.begin(), pv.end(), -1);
            fill(pe.begin(), pe.end(), -1);
            fill(vis.begin(), vis.end(), false);
            struct Q {
                Cost key;
                int u;
                bool operator<(Q o) const { return key > o.key;
                    }
            };
            priority_queue<Q> h;
            dis[s] = 0;
            h.push({0, s});
            while (!h.empty()) {
                int u = h.top().u;
                h.pop();
                if (vis[u]) { continue; }
                vis[u] = true;
                if (u == t) { break; }
                for (int i = 0; i < int(g[u].size()); i++) {
                    auto e = g[u][i];
                    if (vis[e.v] || e.cap == 0) continue;
                    Cost cost = e.cost - dual[e.v] + dual[u];
                    if (dis[e.v] - dis[u] > cost) {
                        dis[e.v] = dis[u] + cost;
                        pv[e.v] = u;
                        pe[e.v] = i;
                        h.push({dis[e.v], e.v});
                    }
                }
            }
            if (!vis[t]) { return false; }
            for (int v = 0; v < n; v++) {
                if (!vis[v]) continue;
                dual[v] -= dis[t] - dis[v];
            }
            return true;
        };
        Flow flow = 0;
        Cost cost = 0, prevCost = -1;
        vector<pair<Flow, Cost>> result;
        result.push_back({flow, cost});
        while (flow < flow_limit) {
            if (!dualRef()) break;
            Flow c = flow_limit - flow;
            for (int v = t; v != s; v = pv[v]) {
                c = min(c, g[pv[v]][pe[v]].cap);
            }
            for (int v = t; v != s; v = pv[v]) {
```

```
                auto& e = g[pv[v]][pe[v]];
                e.cap -= c;
                g[v][e.rev].cap += c;
            }
            Cost d = -dual[s];
            flow += c;
            cost += c * d;
            if (prevCost == d) { result.pop_back(); }
            result.push_back({flow, cost});
            prevCost = cost;
        }
        return result;
    }
private:
    int n;
    struct _edge {
        int v, rev;
        Flow cap;
        Cost cost;
    };
    vector<pair<int, int>> pos;
    vector<vector<_edge>> g;
};
```

2.3 GomoryHu Tree

```
auto gomory(int n, vector<array<int, 3>> e) {
    Flow<int, int> mf(n);
    for (auto [u, v, c] : e) { mf.addEdge(u, v, c, c); }
    vector<array<int, 3>> res;
    vector<int> p(n);
    for (int i = 1; i < n; i++) {
        for (int j = 0; j < int(e.size()); j++) { mf.e[j << 1].cap = mf.e[j << 1 | 1].cap = e[j][2]; }
        int f = mf.maxFlow(i, p[i]);
        auto cut = mf.minCut();
        for (int j = i + 1; j < n; j++) { if (cut[i] == cut[j] && p[i] == p[j]) { p[j] = i; }}
        res.push_back({f, i, p[i]});
    }
    return res;
}
```

2.4 Global Minimum Cut

```
// O(V ^ 3)
template <typename F>
struct GlobalMinCut {
    static constexpr int INF = numeric_limits<F>::max() / 2;
    int n;
    vector<int> vis, wei;
    vector<vector<int>> adj;
    GlobalMinCut(int n) : n(n), vis(n), wei(n), adj(n, vector<int>(n)) {}
    void addEdge(int u, int v, int w){
        adj[u][v] += w;
        adj[v][u] += w;
    }
    int solve() {
        int sz = n;
        int res = INF, x = -1, y = -1;
        auto search = [&]() {
            fill(vis.begin(), vis.begin() + sz, 0);
            fill(wei.begin(), wei.begin() + sz, 0);
            x = y = -1;
            int mx, cur;
            for (int i = 0; i < sz; i++) {
                mx = -1, cur = 0;
                for (int j = 0; j < sz; j++) {
                    if (wei[j] > mx) {
                        mx = wei[j], cur = j;
                    }
                }
                vis[cur] = 1, wei[cur] = -1;
                x = y;
                y = cur;
                for (int j = 0; j < sz; j++) {
                    if (!vis[j]) {
                        wei[j] += adj[cur][j];
                    }
                }
            }
            return mx;
        };
        while (sz > 1) {
            res = min(res, search());
            for (int i = 0; i < sz; i++) {
                adj[x][i] += adj[y][i];
                adj[i][x] = adj[x][i];
            }
            for (int i = 0; i < sz; i++) {
                adj[y][i] = adj[sz - 1][i];
                adj[i][y] = adj[i][sz - 1];
            }
            sz--;
        }
        return res;
    }
};
```

2.5 Bipartite Matching

```
struct BipartiteMatching {
    int n, m;
    vector<vector<int>> adj;
    vector<int> l, r, dis, cur;
    BipartiteMatching(int n, int m) : n(n), m(m), adj(n), l(n, -1), r(m, -1), dis(n), cur(n) {}
    // come on, you know how to write this
    void addEdge(int u, int v) { adj[u].push_back(v); }
    void bfs() {}
    bool dfs(int u) {}
    int maxMatching() {}
    auto minVertexCover() {
        vector<int> L, R;
        for (int u = 0; u < n; u++) {
            if (dis[u] == -1) {
                L.push_back(u);
            } else if (l[u] != -1) {
                R.push_back(l[u]);
            }
        }
        return pair(L, R);
    }
};
```

2.6 GeneralMatching

```
struct GeneralMatching {
    int n;
    vector<vector<int>> adj;
    vector<int> match;
    GeneralMatching(int n) : n(n), adj(n), match(n, -1) {}
    void addEdge(int u, int v) {
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    int maxMatching() {
        vector<int> vis(n), link(n), f(n), dep(n);
        auto find = [&](int u) {
            while (f[u] != u) { u = f[u] = f[f[u]]; }
            return u;
        };
        auto lca = [&](int u, int v) {
            u = find(u);
            v = find(v);
            while (u != v) {
                if (dep[u] < dep[v]) { swap(u, v); }
                u = find(link[match[u]]);
            }
            return u;
        };
        queue<int> q;
        auto blossom = [&](int u, int v, int p) {
            while (find(u) != p) {
                link[u] = v;
                v = match[u];
                if (vis[v] == 0) {
                    vis[v] = 1;
                    q.push(v);
                }
                f[u] = f[v] = p;
                u = link[v];
            }
        };
        auto augment = [&](int u) {
            while (!q.empty()) { q.pop(); }
            iota(f.begin(), f.end(), 0);
            fill(vis.begin(), vis.end(), -1);
            q.push(u), vis[u] = 1, dep[u] = 0;
            while (!q.empty()){
                int u = q.front();
                q.pop();
                for (auto v : adj[u]) {
                    if (vis[v] == -1) {
```

```
                        vis[v] = 0;
                        link[v] = u;
                        dep[v] = dep[u] + 1;
                        if (match[v] == -1) {
                            for (int x = v, y = u, tmp; y != -1; x = tmp, y = x == -1 ? -1 : link[x]) {
                                tmp = match[y], match[x] = y, match[y] = x;
                            }
                            return true;
                        }
                        q.push(match[v]), vis[match[v]] = 1, dep[match[v]] = dep[u] + 2;
                    } else if (vis[v] == 1 && find(v) != find(u)) {
                        int p = lca(u, v);
                        blossom(u, v, p), blossom(v, u, p);
                    }
                }
            }
            return false;
        };
        int res = 0;
        for (int u = 0; u < n; ++u) { if (match[u] == -1) { res += augment(u); } }
        return res;
    }
};
```

2.7 Kuhn Munkres

```
// need perfect matching or not : w intialize with -INF / 0
template <typename Cost>
struct KM {
    static constexpr Cost INF = numeric_limits<Cost>::max() / 2;
    int n;
    vector<Cost> hl, hr, slk;
    vector<int> l, r, pre, vl, vr;
    queue<int> q;
    vector<vector<Cost>> w;
    KM(int n) : n(n), hl(n), hr(n), slk(n), l(n, -1), r(n, -1), pre(n), vl(n), vr(n),
        w(n, vector<Cost>(n, -INF)) {}
    bool check(int x) {
        vl[x] = true;
        if (l[x] != -1) {
            q.push(l[x]);
            return vr[l[x]] = true;
        }
        while (x != -1) { swap(x, r[l[x] = pre[x]]); }
        return false;
    }
    void bfs(int s) {
        fill(slk.begin(), slk.end(), INF);
        fill(vl.begin(), vl.end(), false);
        fill(vr.begin(), vr.end(), false);
        q = {};
        q.push(s);
        vr[s] = true;
        while (true) {
            Cost d;
            while (!q.empty()) {
                int y = q.front();
                q.pop();
                for (int x = 0; x < n; ++x) {
                    if (!vl[x] && slk[x] >= (d = hl[x] + hr[y] - w[x][y])) {
                        pre[x] = y;
                        if (d != 0) {
                            slk[x] = d;
                        } else if (!check(x)) {
                            return;
                        }
                    }
                }
            }
            d = INF;
            for (int x = 0; x < n; ++x) { if (!vl[x] && d > slk[x]) { d = slk[x]; }}
            for (int x = 0; x < n; ++x) {
                if (vl[x]) {
                    hl[x] += d;
                } else {
                    slk[x] -= d;
                }
                if (vr[x]) { hr[x] -= d; }
            }
            for (int x = 0; x < n; ++x) { if (!vl[x] && !slk[x] && !check(x)) { return; }}
        }
    }
    void addEdge(int u, int v, Cost x) { w[u][v] = max(w[u][v], x); }
    Cost solve() {
        for (int i = 0; i < n; ++i) { hl[i] = *max_element(w[i].begin(), w[i].end()); }
        for (int i = 0; i < n; ++i) { bfs(i); }
        Cost res = 0;
        for (int i = 0; i < n; ++i) { res += w[i][l[i]]; }
        return res;
    }
};
```

2.8 Flow Models

- Maximum/Minimum flow with lower bound / Circulation problem
  1. Construct super source S and sink T.
  2. For each edge (x, y, l, u), connect $x \to y$ with capacity $u - l$.
  3. For each vertex v, denote by $in(v)$ the difference between the sum of incoming lower bounds and the sum of outgoing lower bounds.
  4. If $in(v) > 0$, connect $S \to v$ with capacity $in(v)$, otherwise, connect $v \to T$ with capacity $-in(v)$.
     - To maximize, connect $t \to s$ with capacity ∞ (skip this in circulation problem), and let f be the maximum flow from S to T. If $f \neq \sum_{v \in V, in(v)>0} in(v)$, there's no solution. Otherwise, the maximum flow from s to t is the answer.
     - To minimize, let f be the maximum flow from S to T. Connect $t \to s$ with capacity ∞ and let the flow from S to T be f'. If $f + f' \neq \sum_{v \in V, in(v)>0} in(v)$, there's no solution. Otherwise, f' is the answer.
  5. The solution of each edge e is $l_e + f_e$, where f_e corresponds to the flow of edge e on the graph.
- Maximum density induced subgraph
  1. Binary search on answer, suppose we're checking answer T
  2. Construct a max flow model, let K be the sum of all weights
  3. Connect source $s \to v$, $v \in G$ with capacity K
  4. For each edge (u, v, w) in G, connect $u \to v$ and $v \to u$ with capacity w
  5. For $v \in G$, connect it with sink $v \to t$ with capacity $K + 2T - (\sum_{e \in E(v)} w(e)) - 2w(v)$
  6. T is a valid answer if the maximum flow $f < K|V|$
- 0/1 quadratic programming

  $$\sum_x c_x x + \sum_y c_y \bar{y} + \sum_{xy} c_{xy} x \bar{y} + \sum_{xyx'y'} c_{xyx'y'} (x\bar{y} + x'\bar{y'})$$

  can be minimized by the mincut of the following graph:

  1. Create edge (x, t) with capacity c_x and create edge (s, y) with capacity c_y.
  2. Create edge (x, y) with capacity c_{xy}.
  3. Create edge (x, y) and edge (x', y') with capacity $c_{xyx'y'}$.

3 Data Structure

3.1 <ext/pbds>

```
#include <bits/extc++.h>
#include <ext/rope>
using namespace __gnu_pbds;
using namespace __gnu_cxx;
#include <ext/pb_ds/assoc_container.hpp>
typedef tree<int, null_type, std::less<int>, rb_tree_tag,
    tree_order_statistics_node_update> tree_set;
typedef cc_hash_table<int, int> umap;
typedef priority_queue<int> heap;

int main() {
  // rb tree
  tree_set s;
  s.insert(71); s.insert(22);
  assert(*s.find_by_order(0) == 22); assert(*s.find_by_order(1) == 71);
  assert(s.order_of_key(22) == 0); assert(s.order_of_key(71) == 1);
  s.erase(22);
  assert(*s.find_by_order(0) == 71); assert(s.order_of_key(71) == 0);
  // mergable heap
  heap a, b; a.join(b);
  // persistant
  rope<char> r[2];
```

```
    r[1] = r[0];
    std::string st = "abc";
    r[1].insert(0, st.c_str());
    r[1].erase(1, 1);
    std::cout << r[1].substr(0, 2) << std::endl;
    return 0;
}
```

3.2 Li Chao Tree

```
// edu13F MLE with non-deleted pointers
// [) interval because of negative numbers
constexpr i64 INF64 = 4e18;
struct Line {
    i64 a = -INF64, b = -INF64;
    i64 operator()(i64 x) const {
        if (a == -INF64 && b == -INF64) {
            return -INF64;
        } else {
            return a * x + b;
        }
    }
};
constexpr int INF32 = 1e9;
struct LiChao {
    static constexpr int N = 5e6;
    array<Line, N> st;
    array<int, N> lc, rc;
    int n = 0;
    void clear() { n = 0; node(); }
    int node() {
        st[n] = {};
        lc[n] = rc[n] = -1;
        return n++;
    }
    void add(int id, int l, int r, Line line) {
        int m = (l + r) / 2;
        bool lcp = st[id](l) < line(l);
        bool mcp = st[id](m) < line(m);
        if (mcp) { swap(st[id], line); }
        if (r - l == 1) { return; }
        if (lcp != mcp) {
            if (lc[id] == -1) {
                lc[id] = node();
            }
            add(lc[id], l, m, line);
        } else {
            if (rc[id] == -1) {
                rc[id] = node();
            }
            add(rc[id], m, r, line);
        }
    }
    void add(Line line, int l = -INF32 - 1, int r = INF32 + 1) {
        add(0, l, r, line);
    }
    i64 query(int id, int l, int r, i64 x) {
        i64 res = st[id](x);
        if (r - l == 1) { return res; }
        int m = (l + r) / 2;
        if (x < m && lc[id] != -1) {
            res = max(res, query(lc[id], l, m, x));
        } else if (x >= m && rc[id] != -1) {
            res = max(res, query(rc[id], m, r, x));
        }
        return res;
    }
    i64 query(i64 x, int l = -INF32 - 1, int r = INF32 + 1) {
        return query(0, l, r, x);
    }
};
```

3.3 Link-Cut Tree

```
struct Splay {
    array<Splay*, 2> ch = {nullptr, nullptr};
    Splay* fa = nullptr;
    int sz = 1;
    bool rev = false;
    Splay() {}
    void applyRev(bool x) {
        if (x) {
            swap(ch[0], ch[1]);
            rev ^= 1;
        }
    }
    void push() {
        for (auto k : ch) {
            if (k) {
                k->applyRev(rev);
            }
        }
        rev = false;
    }
    void pull() {
        sz = 1;
        for (auto k : ch) {
            if (k) {
            }
        }
    }
    int relation() { return this == fa->ch[1]; }
    bool isRoot() { return !fa || fa->ch[0] != this && fa->ch[1] != this; }
    void rotate() {
        Splay *p = fa;
        bool x = !relation();
        p->ch[!x] = ch[x];
        if (ch[x]) { ch[x]->fa = p; }
        fa = p->fa;
        if (!p->isRoot()) { p->fa->ch[p->relation()] = this; }
        ch[x] = p;
        p->fa = this;
        p->pull();
    }
    void splay() {
        vector<Splay*> s;
        for (Splay *p = this; !p->isRoot(); p = p->fa) { s.push_back(p->fa); }
        while (!s.empty()) {
            s.back()->push();
            s.pop_back();
        }
        push();
        while (!isRoot()) {
            if (!fa->isRoot()) {
                if (relation() == fa->relation()) {
                    fa->rotate();
                } else {
                    rotate();
                }
            }
            rotate();
        }
        pull();
    }
    void access() {
        for (Splay *p = this, *q = nullptr; p; q = p, p = p->fa) {
            p->splay();
            p->ch[1] = q;
            p->pull();
        }
        splay();
    }
    void makeRoot() {
        access();
        applyRev(true);
    }
    Splay* findRoot() {
        access();
        Splay *p = this;
        while (p->ch[0]) { p = p->ch[0]; }
        p->splay();
        return p;
    }
    friend void split(Splay *x, Splay *y) {
        x->makeRoot();
        y->access();
    }
    // link if not connected
    friend void link(Splay *x, Splay *y) {
        x->makeRoot();
        if (y->findRoot() != x) {
            x->fa = y;
        }
    }
    // delete edge if doesn't exist
    friend void cut(Splay *x, Splay *y) {
        split(x, y);
        if (x->fa == y && !x->ch[1]) {
            x->fa = y->ch[0] = nullptr;
```

```
            x->pull();
        }
    }
    bool connected(Splay *x, Splay *y) {
        return x->findRoot() == y->findRoot();
    }
};
```

4 Graph

4.1 2-Edge-Connected Components

```
struct EBCC {
    int n, cnt = 0, T = 0;
    vector<vector<int>> adj, comps;
    vector<int> stk, dfn, low, id;
    EBCC(int n) : n(n), adj(n), dfn(n, -1), low(n), id(n, -1)
        {}
    void addEdge(int u, int v) { adj[u].push_back(v), adj[v].
        push_back(u); }
    void build() { for (int i = 0; i < n; i++) { if (dfn[i] ==
        -1) { dfs(i, -1); }}}
    void dfs(int u, int p) {
        dfn[u] = low[u] = T++;
        stk.push_back(u);
        for (auto v : adj[u]) {
            if (v == p) { continue; }
            if (dfn[v] == -1) {
                dfs(v, u);
                low[u] = min(low[u], low[v]);
            } else if (id[v] == -1) {
                low[u] = min(low[u], dfn[v]);
            }
        }
        if (dfn[u] == low[u]) {
            int x;
            comps.emplace_back();
            do {
                x = stk.back();
                comps.back().push_back(x);
                id[x] = cnt;
                stk.pop_back();
            } while (x != u);
            cnt++;
        }
    }
};
```

4.2 3-Edge-Connected Components

```
// DSU
struct ETCC {
    int n, cnt = 0;
    vector<vector<int>> adj, comps;
    vector<int> in, out, low, up, nx, id;
    ETCC(int n) : n(n), adj(n), in(n, -1), out(in), low(n), up(
        n), nx(in), id(in) {}
    void addEdge(int u, int v) {
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    void build() {
        int T = 0;
        DSU d(n);
        auto merge = [&](int u, int v) {
            d.join(u, v);
            up[u] += up[v];
        };
        auto dfs = [&](auto dfs, int u, int p) -> void {
            in[u] = low[u] = T++;
            for (auto v : adj[u]) {
                if (v == u) { continue; }
                if (v == p) {
                    p = -1;
                    continue;
                }
                if (in[v] == -1) {
                    dfs(dfs, v, u);
                    if (nx[v] == -1 && up[v] <= 1) {
                        up[u] += up[v];
                        low[u] = min(low[u], low[v]);
                        continue;
                    }
                    if (up[v] == 0) { v = nx[v]; }
                    if (low[u] > low[v]) { low[u] = low[v],
                        swap(nx[u], v); }
                    while (v != -1) { merge(u, v); v = nx[v]; }
                } else if (in[v] < in[u]) {
                    low[u] = min(low[u], in[v]);
                    up[u]++;
                } else {
                    for (int &x = nx[u]; x != -1 && in[x] <= in
                        [v] && in[v] < out[x]; x = nx[x]) {
                       merge(u, x);
                    }
                    up[u]--;
                }
            }
            out[u] = T;
        };
        for (int i = 0; i < n; i++) { if (in[i] == -1) { dfs(
            dfs, i, -1); }}
        for (int i = 0; i < n; i++) { if (d.find(i) == i) { id[
            i] = cnt++; }}
        comps.resize(cnt);
        for (int i = 0; i < n; i++) { comps[id[d.find(i)]].
            push_back(i); }
    }
};
```

4.3 Heavy-Light Decomposition

```
struct HLD {
    int n, cur = 0;
    vector<int> sz, top, dep, par, tin, tout, seq;
    vector<vector<int>> adj;
    HLD(int n) : n(n), sz(n, 1), top(n), dep(n), par(n), tin(n)
        , tout(n), seq(n), adj(n) {}
    void addEdge(int u, int v) { adj[u].push_back(v), adj[v].
        push_back(u); }
    void build(int root = 0) {
        top[root] = root, dep[root] = 0, par[root] = -1;
        dfs1(root), dfs2(root);
    }
    void dfs1(int u) {
        if (auto it = find(adj[u].begin(), adj[u].end(), par[u
            ]); it != adj[u].end()) {
            adj[u].erase(it);
        }
        for (auto &v : adj[u]) {
            par[v] = u;
            dep[v] = dep[u] + 1;
            dfs1(v);
            sz[u] += sz[v];
            if (sz[v] > sz[adj[u][0]]) { swap(v, adj[u][0]); }
        }
    }
    void dfs2(int u) {
        tin[u] = cur++;
        seq[tin[u]] = u;
        for (auto v : adj[u]) {
            top[v] = v == adj[u][0] ? top[u] : v;
            dfs2(v);
        }
        tout[u] = cur - 1;
    }
    int lca(int u, int v) {
        while (top[u] != top[v]) {
            if (dep[top[u]] > dep[top[v]]) {
                u = par[top[u]];
            } else {
                v = par[top[v]];
            }
        }
        return dep[u] < dep[v] ? u : v;
    }
    int dist(int u, int v) { return dep[u] + dep[v] - 2 * dep[
        lca(u, v)]; }
    int jump(int u, int k) {
        if (dep[u] < k) { return -1; }
        int d = dep[u] - k;
        while (dep[top[u]] > d) { u = par[top[u]]; }
        return seq[tin[u] - dep[u] + d];
    }
    // u is v's ancestor
    bool isAncestor(int u, int v) { return tin[u] <= tin[v] &&
        tin[v] <= tout[u]; }
    // root's parent is itself
    int rootedParent(int r, int u) {
        if (r == u) { return u; }
```

```
        if (isAncestor(r, u)) { return par[u]; }
        auto it = upper_bound(adj[u].begin(), adj[u].end(), r,
             [&](int x, int y) {
            return tin[x] < tin[y];
        }) - 1;
        return *it;
    }
    // rooted at u, v's subtree size
    int rootedSize(int r, int u) {
        if (r == u) { return n; }
        if (isAncestor(u, r)) { return sz[u]; }
        return n - sz[rootedParent(r, u)];
    }
    int rootedLca(int r, int a, int b) { return lca(a, b) ^ lca
         (a, r) ^ lca(b, r); }
};
```

4.4 Centroid Decomposition

```
vector<int> sz(n), vis(n);
auto dfs1 = [&](auto dfs1, int u, int p) -> void {
    sz[u] = 1;
    for (auto v : g[u]) {
        if (v != p && !vis[v]) {
            dfs1(dfs1, v, u);
            sz[u] += sz[v];
        }
    }
};
auto dfs2 = [&](auto dfs2, int u, int p, int tot) -> int {
    for (auto v : g[u]) {
        if (v != p && !vis[v] && 2 * sz[v] > tot) {
            return dfs2(dfs2, v, u, tot);
        }
    }
    return u;
};

auto dfs = [&](auto dfs, int cen) -> void {
    dfs1(dfs1, cen, -1);
    cen = dfs2(dfs2, cen, -1, sz[cen]);
    vis[cen] = 1;
    dfs1(dfs1, cen, -1);

    for (auto v : g[cen]) {
        if (!vis[v]) {
            dfs(dfs, v);
        }
    }
};
dfs(dfs, 0);
```

4.5 Strongly Connected Components

```
struct SCC {
    int n, cnt = 0, cur = 0;
    vector<int> id, dfn, low, stk;
    vector<vector<int>> adj, comps;
    void addEdge(int u, int v) { adj[u].push_back(v); }
    SCC(int n) : n(n), id(n, -1), dfn(n, -1), low(n, -1), adj(n
        ) {}
    void build() {
        auto dfs = [&](auto dfs, int u) -> void {
            dfn[u] = low[u] = cur++;
            stk.push_back(u);
            for (auto v : adj[u]) {
                if (dfn[v] == -1) {
                    dfs(dfs, v);
                    low[u] = min(low[u], low[v]);
                } else if (id[v] == -1) {
                    low[u] = min(low[u], dfn[v]);
                }
            }
            if (dfn[u] == low[u]) {
                int v;
                comps.emplace_back();
                do {
                    v = stk.back();
                    comps.back().push_back(v);
                    id[v] = cnt;
                    stk.pop_back();
                } while (u != v);
                cnt++;
            }
        };
        for (int i = 0; i < n; i++) { if (dfn[i] == -1) { dfs(
             dfs, i); }}
        for (int i = 0; i < n; i++) { id[i] = cnt - 1 - id[i];
             }
        reverse(comps.begin(), comps.end());
    }
    // the comps are in topological sorted order
};
```

4.6 2-SAT

```
struct TwoSat {
    int n, N;
    vector<vector<int>> adj;
    vector<int> ans;
    TwoSat(int n) : n(n), N(n), adj(2 * n) {}
    // u == x
    void addClause(int u, bool x) { adj[2 * u + !x].push_back(2
         * u + x); }
    // u == x || v == y
    void addClause(int u, bool x, int v, bool y) {
        adj[2 * u + !x].push_back(2 * v + y);
        adj[2 * v + !y].push_back(2 * u + x);
    }
    // u == x -> v == y
    void addImply(int u, bool x, int v, bool y) { addClause(u,
         !x, v, y); }
    void addVar() {
        adj.emplace_back(), adj.emplace_back();
        N++;
    }
    // at most one in var is true
    // adds prefix or as supplementary variables
    void atMostOne(const vector<pair<int, bool>> &vars) {
        int sz = vars.size();
        for (int i = 0; i < sz; i++) {
            addVar();
            auto [u, x] = vars[i];
            addImply(u, x, N - 1, true);
            if (i > 0) {
                addImply(N - 2, true, N - 1, true);
                addClause(u, !x, N - 2, false);
            }
        }
    }
    // does not return supplementary variables from atMostOne()
    bool satisfiable() {
        // run tarjan scc on 2 * N
        for (int i = 0; i < 2 * N; i++) { if (dfn[i] == -1) {
             dfs(dfs, i); }}
        for (int i = 0; i < N; i++) { if (id[2 * i] == id[2 * i
             + 1]) { return false; }}
        ans.resize(n);
        for (int i = 0; i < n; i++) { ans[i] = id[2 * i] > id[2
             * i + 1]; }
        return true;
    }
};
```

4.7 count 3-cycles and 4-cycles

```
sort(ord.begin(), ord.end(), [&](auto i, auto j) { return pair(
     deg[i], i) > pair(deg[j], j); });
for (int i = 0; i < n; i++) { rnk[ord[i]] = i; }
if (rnk[u] < rnk[v]) { dag[u].push_back(v); }
// c3
for (int x = 0; x < n; x++) {
    for (auto y : dag[x]) {
        vis[y] = 1;
    }
    for (auto y : dag[x]) {
        for (auto z : dag[y]) {
            ans += vis[z];
        }
    }
    for (auto y : dag[x]) {
        vis[y] = 0;
    }
}
// c4
for (int x = 0; x < n; x++) {
    for (auto y : dag[x]) {
        for (auto z : adj[y]) {
            if (rnk[z] > rnk[x]) {
                ans += vis[z]++;
            }
        }
    }
    for (auto y : dag[x]) {
```

```
        for (auto z : adj[y]) {
            if (rnk[z] > rnk[x]) {
                vis[z]--;
            }
        }
    }
}
```

4.8 Minimum Mean Cycle

create a new vertex S, connect S to all vertices with arbitrary weight (0). Let $f_i(u)$ be the shortest path from S to u with exactly i edges.

$$ans = \min_{f_{n+1}(i) != \infty} \max_{j=1}^{n} \frac{f_{n+1}(i) - f_j(i)}{n + 1 - j}$$

4.9 Directed Minimum Spanning Tree

```
// DSU with rollback
template <typename Cost>
struct DMST {
    int n;
    vector<int> s, t, lc, rc, h;
    vector<Cost> c, tag;
    DMST(int n) : n(n), h(n, -1) {}
    void addEdge(int u, int v, Cost w) {
        int id = s.size();
        s.push_back(u), t.push_back(v), c.push_back(w);
        lc.push_back(-1), rc.push_back(-1);
        tag.emplace_back();
        h[v] = merge(h[v], id);
    }
    pair<Cost, vector<int>> build(int root = 0) {
        DSU d(n);
        Cost res{};
        vector<int> vis(n, -1), path(n), q(n), in(n, -1);
        vis[root] = root;
        vector<pair<int, vector<int>>> cycles;
        for (auto r = 0; r < n; ++r) {
            auto u = r, b = 0, w = -1;
            while (!~vis[u]) {
                if (!~h[u]) { return {-1, {}}; }
                push(h[u]);
                int e = h[u];
                res += c[e], tag[h[u]] -= c[e];
                h[u] = pop(h[u]);
                q[b] = e, path[b++] = u, vis[u] = r;
                u = d.find(s[e]);
                if (vis[u] == r) {
                    int cycle = -1, e = b;
                    do {
                        w = path[--b];
                        cycle = merge(cycle, h[w]);
                    } while (d.join(u, w));
                    u = d.find(u);
                    h[u] = cycle, vis[u] = -1;
                    cycles.emplace_back(u, vector<int>(q.begin
                        () + b, q.begin() + e));
                }
            }
            for (auto i = 0; i < b; ++i) { in[d.find(t[q[i]])]
                = q[i]; }
        }
        reverse(cycles.begin(), cycles.end());
        for (const auto &[u, comp] : cycles) {
            int count = int(comp.size()) - 1;
            d.back(count);
            int ine = in[u];
            for (auto e : comp) { in[d.find(t[e])] = e; }
            in[d.find(t[ine])] = ine;
        }
        vector<int> par;
        par.reserve(n);
        for (auto i : in) { par.push_back(i != -1 ? s[i] : -1);
            }
        return {res, par};
    }
    void push(int u) {
        c[u] += tag[u];
        if (int l = lc[u]; l != -1) { tag[l] += tag[u]; }
        if (int r = rc[u]; r != -1) { tag[r] += tag[u]; }
        tag[u] = 0;
    }
    int merge(int u, int v) {
        if (u == -1 || v == -1) { return u != -1 ? u : v; }
        push(u);
        push(v);
        if (c[u] > c[v]) { swap(u, v); }
        rc[u] = merge(v, rc[u]);
        swap(lc[u], rc[u]);
        return u;
    }
    int pop(int u) {
        push(u);
        return merge(lc[u], rc[u]);
    }
};
```

4.10 Maximum Clique

```
pair<int, vector<int>> maxClique(const vector<bitset<N>> adj) {
    int n = adj.size();
    int mx = 0;
    vector<int> ans, cur;
    auto rec = [&](auto rec, bitset<N> s) -> void {
        int sz = s.count();
        if (int(cur.size()) > mx) { mx = cur.size(), ans = cur;
            }
        if (int(cur.size()) + sz <= mx) { return; }
        int e1 = -1, e2 = -1;
        vector<int> d(n);
        for (int i = 0; i < n; i++) {
            if (s[i]) {
                d[i] = (adj[i] & s).count();
                if (e1 == -1 || d[i] > d[e1]) { e1 = i; }
                if (e2 == -1 || d[i] < d[e2]) { e2 = i; }
            }
        }
        if (d[e1] >= sz - 2) {
            cur.push_back(e1);
            auto s1 = adj[e1] & s;
            rec(rec, s1);
            cur.pop_back();
            return;
        }
        cur.push_back(e2);
        auto s2 = adj[e2] & s;
        rec(rec, s2);
        cur.pop_back();
        s.reset(e2);
        rec(rec, s);
    };
    bitset<N> all;
    for (int i = 0; i < n; i++) {
        all.set(i);
    }
    rec(rec, all);
    return pair(mx, ans);
}
```

4.11 Dominator Tree

```
// res : parent of each vertex in dominator tree, -1 is root,
    -2 if not in tree
struct DominatorTree {
    int n, cur = 0;
    vector<int> dfn, rev, fa, sdom, dom, val, rp, res;
    vector<vector<int>> adj, rdom, r;
    DominatorTree(int n) : n(n), dfn(n, -1), res(n, -2), adj(n)
        , rdom(n), r(n) {
        rev = fa = sdom = dom = val = rp = dfn;
    }
    void addEdge(int u, int v) {
        adj[u].push_back(v);
    }
    void dfs(int u) {
        dfn[u] = cur;
        rev[cur] = u;
        fa[cur] = sdom[cur] = val[cur] = cur;
        cur++;
        for (int v : adj[u]) {
            if (dfn[v] == -1) {
                dfs(v);
                rp[dfn[v]] = dfn[u];
            }
            r[dfn[v]].push_back(dfn[u]);
        }
    }
    int find(int u, int c) {
        if (fa[u] == u) { return c != 0 ? -1 : u; }
        int p = find(fa[u], 1);
        if (p == -1) { return c != 0 ? fa[u] : val[u]; }
```

```
        if (sdom[val[u]] > sdom[val[fa[u]]]) { val[u] = val[fa[
            u]]; }
        fa[u] = p;
        return c != 0 ? p : val[u];
    }
    void build(int s = 0) {
        dfs(s);
        for (int i = cur - 1; i >= 0; i--) {
            for (int u : r[i]) { sdom[i] = min(sdom[i], sdom[
                find(u, 0)]); }
            if (i > 0) { rdom[sdom[i]].push_back(i); }
            for (int u : rdom[i]) {
                int p = find(u, 0);
                if (sdom[p] == i) {
                    dom[u] = i;
                } else {
                    dom[u] = p;
                }
            }
            if (i > 0) { fa[i] = rp[i]; }
        }
        res[s] = -1;
        for (int i = 1; i < cur; i++) { if (sdom[i] != dom[i])
            { dom[i] = dom[dom[i]]; }}
        for (int i = 1; i < cur; i++) { res[rev[i]] = rev[dom[i
            ]]; }
    }
};
```

4.12 Vizing's Theorem

```
// bipartite
e[i] = pair(u, v + a), deg[u]++, deg[v + a]++;
int col = *max_element(deg.begin(), deg.end());
vector<int> ans(m, -1);
vector has(a + b, vector<pair<int, int>>(col, {-1, -1}));
for (int i = 0; i < m; i++) {
    auto [u, v] = e[i];
    vector<int> c;
    for (auto x : {u, v}) {
        c.push_back(0);
        while (has[x][c.back()].first != -1) { c.back()++; }
    }
    if (c[0] != c[1]) {
        auto dfs = [&](auto dfs, int u, int x) -> void {
            auto [v, i] = has[u][c[x]];
            if (v != -1) {
                if (has[v][c[x ^ 1]].first != -1) {
                    dfs(dfs, v, x ^ 1);
                } else {
                    has[v][c[x]] = {-1, -1};
                }
                has[u][c[x ^ 1]] = {v, i}, has[v][c[x ^ 1]] = {
                    u, i};
                ans[i] = c[x ^ 1];
            }
        };
        dfs(dfs, v, 0);
    }
    has[u][c[0]] = {v, i};
    has[v][c[0]] = {u, i};
    ans[i] = c[0];
}
// general
auto vizing(int n, const vector<pair<int, int>> &e) {
    vector<int> deg(n);
    for (auto [u, v] : e) {
        deg[u]++, deg[v]++;
    }
    int col = *max_element(deg.begin(), deg.end()) + 1;
    vector<int> free(n);
    vector ans(n, vector<int>(n, -1));
    vector at(n, vector<int>(col, -1));
    auto update = [&](int u) {
        free[u] = 0;
        while (at[u][free[u]] != -1) {
            free[u]++;
        }
    };
    auto color = [&](int u, int v, int c1) {
        int c2 = ans[u][v];
        ans[u][v] = ans[v][u] = c1;
        at[u][c1] = v, at[v][c1] = u;
        if (c2 != -1) {
            at[u][c2] = at[v][c2] = -1;
            free[u] = free[v] = c2;
        } else {
            update(u), update(v);
        }
        return c2;
    };
    auto flip = [&](int u, int c1, int c2) {
        int v = at[u][c1];
        swap(at[u][c1], at[u][c2]);
        if (v != -1) {
            ans[u][v] = ans[v][u] = c2;
        }
        if (at[u][c1] == -1) {
            free[u] = c1;
        }
        if (at[u][c2] == -1) {
            free[u] = c2;
        }
        return v;
    };
    for (int i = 0; i < int(e.size()); i++) {
        auto [u, v1] = e[i];
        int v2 = v1, c1 = free[u], c2 = c1, d;
        vector<pair<int, int>> fan;
        vector<int> vis(col);
        while (ans[u][v1] == -1) {
            fan.emplace_back(v2, d = free[v2]);
            if (at[v2][c2] == -1) {
                for (int j = int(fan.size()) - 1; j >= 0; j--)
                    {
                    c2 = color(u, fan[j].first, c2);
                }
            } else if (at[u][d] == -1) {
                for (int j = int(fan.size()) - 1; j >= 0; j--)
                    {
                    color(u, fan[j].first, fan[j].second);
                }
            } else if (vis[d] == 1) {
                break;
            } else {
                vis[d] = 1, v2 = at[u][d];
            }
        }
        if (ans[u][v1] == -1) {
            while (v2 != -1) {
                v2= flip(v2, c2, d);
                swap(c2, d);
            }
            if (at[u][c1] != -1) {
                int j = int(fan.size()) - 2;
                while (j >= 0 && fan[j].second != c2) {
                    j--;
                }
                while (j >= 0) {
                    color(u, fan[j].first, fan[j].second);
                    j--;
                }
            } else {
                i--;
            }
        }
    }
    return pair(col, ans);
}
```

5 String

5.1 Prefix Function

```
template <typename T>
vector<int> prefixFunction(const T &s) {
    int n = int(s.size());
    vector<int> p(n);
    for (int i = 1; i < n; i++) {
        int j = p[i - 1];
        while (j > 0 && s[i] != s[j]) { j = p[j - 1]; }
        if (s[i] == s[j]) { j++; }
        p[i] = j;
    }
    return p;
}
```

5.2 Z Function

```
template <typename T>
vector<int> zFunction(const T &s) {
```

```cpp
    int n = int(s.size());
    if (n == 0) return {};
    vector<int> z(n);
    for (int i = 1, j = 0; i < n; i++) {
        int &k = z[i];
        k = j + z[j] <= i ? 0 : min(j + z[j] - i, z[i - j]);
        while (i + k < n && s[k] == s[i + k]) { k++; }
        if (j + z[j] < i + z[i]) { j = i; }
    }
    z[0] = n;
    return z;
}
```

5.3 Suffix Array

```cpp
// need to discretize
struct SuffixArray {
    int n;
    vector<int> sa, as, ha;
template <typename T>
    vector<int> sais(const T &s) {
        int n = s.size(), m = *max_element(s.begin(), s.end()) + 1;
        vector<int> pos(m + 1), f(n);
        for (auto ch : s) { pos[ch + 1]++; }
        for (int i = 0; i < m; i++) { pos[i + 1] += pos[i]; }
        for (int i = n - 2; i >= 0; i--) { f[i] = s[i] != s[i + 1] ? s[i] < s[i + 1] : f[i + 1]; }
        vector<int> x(m), sa(n);
        auto induce = [&](const vector<int> &ls) {
            fill(sa.begin(), sa.end(), -1);
            auto L = [&](int i) { if (i >= 0 && !f[i]) { sa[x[s[i]]++] = i; }};
            auto S = [&](int i) { if (i >= 0 && f[i]) { sa[--x[s[i]]] = i; }};
            for (int i = 0; i < m; i++) { x[i] = pos[i + 1]; }
            for (int i = int(ls.size()) - 1; i >= 0; i--) { S(ls[i]); }
            for (int i = 0; i < m; i++) { x[i] = pos[i]; }
            L(n - 1);
            for (int i = 0; i < n; i++) { L(sa[i] - 1); }
            for (int i = 0; i < m; i++) { x[i] = pos[i + 1]; }
            for (int i = n - 1; i >= 0; i--) { S(sa[i] - 1); }
        };
        auto ok = [&](int i) { return i == n || !f[i - 1] && f[i]; };
        auto same = [&](int i, int j) {
            do { if (s[i++] != s[j++]) { return false; }} while (!ok(i) && !ok(j));
            return ok(i) && ok(j);
        };
        vector<int> val(n), lms;
        for (int i = 1; i < n; i++) { if (ok(i)) { lms.push_back(i); }}
        induce(lms);
        if (!lms.empty()) {
            int p = -1, w = 0;
            for (auto v : sa) {
                if (v != 0 && ok(v)) {
                    if (p != -1 && same(p, v)) { w--; }
                    val[p = v] = w++;
                }
            }
            auto b = lms;
            for (auto &v : b) { v = val[v]; }
            b = sais(b);
            for (auto &v : b) { v = lms[v]; }
            induce(b);
        }
        return sa;
    }
template <typename T>
    SuffixArray(const T &s) : n(s.size()), sa(sais(s)), as(n), ha(n - 1) {
        for (int i = 0; i < n; i++) { as[sa[i]] = i; }
        for (int i = 0, j = 0; i < n; ++i) {
            if (as[i] == 0) {
                j = 0;
            } else {
                for (j -= j > 0; i + j < n && sa[as[i] - 1] + j < n && s[i + j] == s[sa[as[i] - 1] + j]; ) { ++j; }
                ha[as[i] - 1] = j;
            }
        }
    }
};
```

5.4 Manacher's Algorithm

```cpp
// returns radius of t, length of s : rad(t) - 1, radius of s : rad(t) / 2
vector<int> manacher(string s) {
    string t = "#";
    for (auto c : s) { t += c, t += '#'; }
    int n = t.size();
    vector<int> r(n);
    for (int i = 0, j = 0; i < n; i++) {
        if (2 * j - i >= 0 && j + r[j] > i) { r[i] = min(r[2 * j - i], j + r[j] - i); }
        while (i - r[i] >= 0 && i + r[i] < n && t[i - r[i]] == t[i + r[i]]) { r[i]++; }
        if (i + r[i] > j + r[j]) { j = i; }
    }
    return r;
}
```

5.5 Aho-Corasick Automaton

```cpp
constexpr int K = 26;
struct Node {
    array<int, K> nxt;
    int fail = -1;
    // other vars
    Node() { nxt.fill(-1); }
};
vector<Node> aho(1);
for (int i = 0; i < n; i++) {
    string s;
    cin >> s;
    int u = 0;
    for (auto ch : s) {
        int c = ch - 'a';
        if (aho[u].nxt[c] == -1) {
            aho[u].nxt[c] = aho.size();
            aho.emplace_back();
        }
        u = aho[u].nxt[c];
    }
}
vector<int> q;
for (auto &i : aho[0].nxt) {
    if (i == -1) {
        i = 0;
    } else {
        q.push_back(i);
        aho[i].fail = 0;
    }
}
for (int i = 0; i < int(q.size()); i++) {
    int u = q[i];
    if (u > 0) {
        // maintain
    }
    for (int c = 0; c < K; c++) {
        if (int v = aho[u].nxt[c]; v != -1) {
            aho[v].fail = aho[aho[u].fail].nxt[c];
            q.push_back(v);
        } else {
            aho[u].nxt[c] = aho[aho[u].fail].nxt[c];
        }
    }
}
```

5.6 Suffix Automaton

```cpp
constexpr int K = 26;
struct Node{
    int len = 0, link = -1, cnt = 0;
    array<int, K> nxt;
    Node() { nxt.fill(-1); }
};
vector<Node> sam(1);
auto extend = [&](int c) {
    static int last = 0;
    int p = last, cur = sam.size();
    sam.emplace_back();
    sam[cur].len = sam[p].len + 1;
    sam[cur].cnt = 1;
    while (p != -1 && sam[p].nxt[c] == -1) {
        sam[p].nxt[c] = cur;
        p = sam[p].link;
    }
    if (p == -1) {
        sam[cur].link = 0;
```

```
    } else {
        int q = sam[p].nxt[c];
        if (sam[p].len + 1 == sam[q].len) {
            sam[cur].link = q;
        } else {
            int clone = sam.size();
            sam.emplace_back();
            sam[clone].len = sam[p].len + 1;
            sam[clone].link = sam[q].link;
            sam[clone].nxt = sam[q].nxt;
            while (p != -1 && sam[p].nxt[c] == q) {
                sam[p].nxt[c] = clone;
                p = sam[p].link;
            }
            sam[q].link = sam[cur].link = clone;
        }
    }
    last = cur;
};
for (auto ch : s) {
    extend(ch - 'a');
}
int N = sam.size();
vector<vector<int>> g(N);
for (int i = 1; i < N; i++) {
    g[sam[i].link].push_back(i);
}
```

5.7 Lexicographically Smallest Rotation

```
template <typename T>
T minRotation(T s) {
    int n = s.size();
    int i = 0, j = 1;
    s.insert(s.end(), s.begin(), s.end());
    while (i < n && j < n) {
        int k = 0;
        while (k < n && s[i + k] == s[j + k]) {
            k++;
        }
        if (s[i + k] <= s[j + k]) {
            j += k + 1;
        } else {
            i += k + 1;
        }
        if (i == j) {
            j++;
        }
    }
    int ans = i < n ? i : j;
    return T(s.begin() + ans, s.begin() + ans + n);
}
```

6 Math

6.1 Extended GCD

```
array<i64, 3> extgcd(i64 a, i64 b) {
    if (b == 0) { return {a, 1, 0}; }
    auto [g, x, y] = extgcd(b, a % b);
    return {g, y, x - a / b * y};
}
```

6.2 Chinese Remainder Theorem

```
// returns (rem, mod), n = 0 return (0, 1), no solution return
    (0, 0)
pair<i64, i64> crt(vector<i64> r, vector<i64> m) {
    int n = r.size();
    for (int i = 0; i < n; i++) {
        r[i] %= m[i];
        if (r[i] < 0) { r[i] += m[i]; }
    }
    i64 r0 = 0, m0 = 1;
    for (int i = 0; i < n; i++) {
        i64 r1 = r[i], m1 = m[i];
        if (m0 < m1) { swap(r0, r1), swap(m0, m1); }
        if (m0 % m1 == 0) {
            if (r0 % m1 != r1) { return {0, 0}; }
            continue;
        }
        auto [g, a, b] = extgcd(m0, m1);
        i64 u1 = m1 / g;
        if ((r1 - r0) % g != 0) { return {0, 0}; }
        i64 x = (r1 - r0) / g % u1 * a % u1;
        r0 += x * m0;
        m0 *= u1;
        if (r0 < 0) { r0 += m0; }
    }
    return {r0, m0};
}
```

6.3 NTT and polynomials

```
template <int P>
struct Modint {
    int v;
    constexpr Modint() : v(0) {}
    constexpr Modint(i64 v) : v((v % P + P) % P) {}
    constexpr friend Modint operator+(Modint a, Modint b) {
        return Modint((a.v + b.v) % P); }
    constexpr friend Modint operator-(Modint a, Modint b) {
        return Modint((a.v + P - b.v) % P); }
    constexpr friend Modint operator*(Modint a, Modint b) {
        return Modint(1LL * a.v * b.v % P); }
    constexpr Modint qpow(i64 p) {
        Modint res = 1, x = v;
        while (p > 0) {
            if (p & 1) { res = res * x; }
            x = x * x;
            p >>= 1;
        }
        return res;
    }
    constexpr Modint inv() { return qpow(P - 2); }
};
template<int P>
constexpr Modint<P> findPrimitiveRoot() {
    Modint<P> i = 2;
    int k = __builtin_ctz(P - 1);
    while (true) {
        if (i.qpow((P - 1) / 2).v != 1) { break; }
        i = i + 1;
    }
    return i.qpow(P - 1 >> k);
}
template <int P>
constexpr Modint<P> primitiveRoot = findPrimitiveRoot<P>();
vector<int> rev;
template <int P>
vector<Modint<P>> roots{0, 1};
template <int P>
void dft(vector<Modint<P>> &a) {
    int n = a.size();
    if (n == 1) { return; }
    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) { swap(a[i],
        a[rev[i]]); }}
    if (roots<P>.size() < n) {
        int k = __builtin_ctz(roots<P>.size());
        roots<P>.resize(n);
        while ((1 << k) < n) {
            auto e = Modint<P>(primitiveRoot<P>).qpow(P - 1 >>
                k + 1);
            for (int i = 1 << k - 1; i < 1 << k; i++) {
                roots<P>[2 * i] = roots<P>[i];
                roots<P>[2 * i + 1] = roots<P>[i] * e;
            }
            k++;
        }
    }
    // fft : just do roots[i] = exp(2 * PI / n * i * complex<
        double>(0, 1))
    for (int k = 1; k < n; k *= 2) {
        for (int i = 0; i < n; i += 2 * k) {
            for (int j = 0; j < k; j++) {
                Modint<P> u = a[i + j];
                Modint<P> v = a[i + j + k] * roots<P>[k + j];
                // fft : v = a[i + j + k] * roots[n / (2 * k) *
                    j]
                a[i + j] = u + v;
                a[i + j + k] = u - v;
            }
        }
    }
}
template <int P>
```

```
void idft(vector<Modint<P>> &a) {
    int n = a.size();
    reverse(a.begin() + 1, a.end());
    dft(a);
    Modint<P> x = (1 - P) / n;
    for (int i = 0; i < n; i++) { a[i] = a[i] * x; }
}
template <int P>
struct Poly : vector<Modint<P>> {
    using Mint = Modint<P>;
    Poly() {}
    explicit Poly(int n) : vector<Mint>(n) {}
    explicit Poly(const vector<Mint> &a) : vector<Mint>(a) {}
    explicit Poly(const initializer_list<Mint> &a) : vector<
        Mint>(a) {}
template<class F>
    explicit Poly(int n, F f) : vector<Mint>(n) { for (int i =
        0; i < n; i++) { (*this)[i] = f(i); }}
template<class InputIt>
    explicit constexpr Poly(InputIt first, InputIt last) :
        vector<Mint>(first, last) {}
    Poly mulxk(int k) {
        auto b = *this;
        b.insert(b.begin(), k, 0);
        return b;
    }
    Poly modxk(int k) {
        k = min(k, int(this->size()));
        return Poly(this->begin(), this->begin() + k);
    }
    Poly divxk(int k) {
        if (this->size() <= k) { return Poly(); }
        return Poly(this->begin() + k, this->end());
    }
    friend Poly operator+(const Poly &a, const Poly &b) {
        Poly res(max(a.size(), b.size()));
        for (int i = 0; i < int(a.size()); i++) { res[i] = res[
            i] + a[i]; }
        for (int i = 0; i < int(b.size()); i++) { res[i] = res[
            i] + b[i]; }
        return res;
    }
    friend Poly operatorkj-(const Poly &a, const Poly &b) {
        Poly res(max(a.size(), b.size()));
        for (int i = 0; i < int(a.size()); i++) { res[i] = res[
            i] + a[i]; }
        for (int i = 0; i < int(b.size()); i++) { res[i] = res[
            i] - b[i]; }
        return res;
    }
    friend Poly operator*(Poly a, Poly b) {
        if (a.empty() || b.empty()) { return Poly(); }
        int sz = 1, tot = a.size() + b.size() - 1;
        while (sz < tot) { sz *= 2; }
        a.resize(sz);
        b.resize(sz);
        dft(a);
        dft(b);
        for (int i = 0; i < sz; i++) { a[i] = a[i] * b[i]; }
        idft(a);
        a.resize(tot);
        return a;
    }
    friend Poly operator*(Poly a, Mint b) {
        for (int i = 0; i < int(a.size()); i++) { a[i] = a[i] *
             b; }
        return a;
    }
    Poly derivative()  {
        if (this->empty()) { return Poly(); }
        Poly res(this->size() - 1);
        for (int i = 0; i < this->size() - 1; ++i) { res[i] = (
            i + 1) * (*this)[i + 1]; }
        return res;
    }
    Poly integral() {
        Poly res(this->size() + 1);
        for (int i = 0; i < this->size(); ++i) { res[i + 1] =
            (*this)[i] * Mint(i + 1).inv(); }
        return res;
    }
    Poly inv(int m) {
        // a[0] != 0
        Poly x({(*this)[0].inv()});
        int k = 1;
        while (k < m) {
            k *= 2;
            x = (x * (Poly({2}) - modxk(k) * x)).modxk(k);
        }
        return x.modxk(m);
    }
    Poly log(int m)  {
        return (derivative() * inv(m)).integral().modxk(m);
    }
    Poly exp(int m) {
        Poly x({1});
        int k = 1;
        while (k < m) {
            k *= 2;
            x = (x * (Poly({1}) - x.log(k) + modxk(k))).modxk(k
                );
        }
        return x.modxk(m);
    }
    Poly pow(i64 k, int m) {
        if (k == 0) { return Poly(m, [&](int i) { return i ==
            0; }); }
        int i = 0;
        while (i < this->size() && (*this)[i].v == 0) { i++; }
        if (i == this->size() || __int128(i) * k >= m) { return
             Poly(m); }
        Mint v = (*this)[i];
        auto f = divxk(i) * v.inv();
        return (f.log(m - i * k) * k).exp(m - i * k).mulxk(i *
            k) * v.qpow(k);
    }
    Poly sqrt(int m) {
        // a[0] == 1, otherwise quadratic residue?
        Poly x({1});
        int k = 1;
        while (k < m) {
            k *= 2;
            x = (x + (modxk(k) * x.inv(k)).modxk(k)) * ((P + 1)
                / 2);
        }
        return x.modxk(m);
    }
    Poly mulT(Poly b) const {
        if (b.empty()) { return Poly(); }
        int n = b.size();
        reverse(b.begin(), b.end());
        return (*this * b).divxk(n - 1);
    }
    vector<Mint> evaluate(vector<Mint> x) {
        if (this->empty()) { return vector<Mint>(x.size()); }
        int n = max(x.size(), this->size());
        vector<Poly> q(4 * n);
        vector<Mint> ans(x.size());
        x.resize(n);
        auto build = [&](auto build, int id, int l, int r) ->
            void {
            if (r - l == 1) {
                q[id] = Poly({1, -x[l].v});
            } else {
                int m = (l + r) / 2;
                build(build, 2 * id, l, m);
                build(build, 2 * id + 1, m, r);
                q[id] = q[2 * id] * q[2 * id + 1];
            }
        };
        build(build, 1, 0, n);
        auto work = [&](auto work, int id, int l, int r, const
            Poly &num) -> void {
            if (r - l == 1) {
                if (l < int(ans.size())) { ans[l] = num[0]; }
            } else {
                int m = (l + r) / 2;
                work(work, 2 * id, l, m, num.mulT(q[2 * id +
                    1]).modxk(m - l));
                work(work, 2 * id + 1, m, r, num.mulT(q[2 * id
                    ]).modxk(r - m));
            }
        };
        work(work, 1, 0, n, mulT(q[1].inv(n)));
        return ans;
    }
};
template <int P>
Poly<P> interpolate(vector<Modint<P>> x, vector<Modint<P>> y) {
    // f(xi) = yi
    int n = x.size();
    vector<Poly<P>> p(4 * n), q(4 * n);
    auto dfs1 = [&](auto dfs1, int id, int l, int r) -> void {
```

```
        if (l == r) {
            p[id] = Poly<P>({-x[l].v, 1});
            return;
        }
        int m = l + r >> 1;
        dfs1(dfs1, id << 1, l, m);
        dfs1(dfs1, id << 1 | 1, m + 1, r);
        p[id] = p[id << 1] * p[id << 1 | 1];
    };
    dfs1(dfs1, 1, 0, n - 1);
    Poly<P> f = Poly<P>(p[1].derivative().evaluate(x));
    auto dfs2 = [&](auto dfs2, int id, int l, int r) -> void {
        if (l == r) {
            q[id] = Poly<P>({y[l] * f[l].inv()});
            return;
        }
        int m = l + r >> 1;
        dfs2(dfs2, id << 1, l, m);
        dfs2(dfs2, id << 1 | 1, m + 1, r);
        q[id] = q[id << 1] * p[id << 1 | 1] + q[id << 1 | 1] * p[id << 1];
    };
    dfs2(dfs2, 1, 0, n - 1);
    return q[1];
}
```

6.4 NTT Prime List

Prime	Root	Prime	Root
7681	17	167772161	3
12289	11	104857601	3
40961	3	985661441	3
65537	3	998244353	3
786433	10	1107296257	10
5767169	3	2013265921	31
7340033	3	2810183681	11
23068673	3	2885681153	3
469762049	3	605028353	3

6.5 Newton's Method

$$Q_{k+1} = Q_k - \frac{F(Q_k)}{F'(Q_k)} \pmod{x^{2^{k+1}}}$$

6.6 Fast Walsh-Hadamard Transform

1. XOR Convolution
   - $f(A) = (f(A_0) + f(A_1), f(A_0) - f(A_1))$
   - $f^{-1}(A) = (f^{-1}(\frac{A_0+A_1}{2}), f^{-1}(\frac{A_0-A_1}{2}))$
2. OR Convolution
   - $f(A) = (f(A_0), f(A_0) + f(A_1))$
   - $f^{-1}(A) = (f^{-1}(A_0), f^{-1}(A_1) - f^{-1}(A_0))$
3. AND Convolution
   - $f(A) = (f(A_0) + f(A_1), f(A_1))$
   - $f^{-1}(A) = (f^{-1}(A_0) - f^{-1}(A_1), f^{-1}(A_1))$

6.7 Simplex Algorithm

Description: maximize $\mathbf{c}^T\mathbf{x}$ subject to $A\mathbf{x} \le \mathbf{b}$ and $\mathbf{x} \ge 0$. Returns $-\infty$ if infeasible and ∞ if unbounded.

```
const double eps = 1e-9;
const double inf = 1e+9;
int n, m;
vector<vector<double>> d;
vector<int> p, q;
void pivot(int r, int s) {
  double inv = 1.0 / d[r][s];
  for (int i = 0; i < m + 2; ++i) {
    for (int j = 0; j < n + 2; ++j) {
      if (i != r && j != s) d[i][j] -= d[r][j] * d[i][s] * inv;
    }
  }
  for (int i = 0; i < m + 2; ++i) if (i != r) d[i][s] *= -inv;
  for (int j = 0; j < n + 2; ++j) if (j != s) d[r][j] *= +inv;
  d[r][s] = inv;
  swap(p[r], q[s]);
}
bool phase(int z) {
  int x = m + z;
  while (true) {
    int s = -1;
    for (int i = 0; i <= n; ++i) {
      if (!z && q[i] == -1) continue;
      if (s == -1 || d[x][i] < d[x][s]) s = i;
    }
    if (d[x][s] > -eps) return true;
    int r = -1;
    for (int i = 0; i < m; ++i) {
      if (d[i][s] < eps) continue;
      if (r == -1 || d[i][n + 1] / d[i][s] < d[r][n + 1] / d[r][s]) r = i;
    }
    if (r == -1) return false;
    pivot(r, s);
  }
}
vector<double> solve(const vector<vector<double>> &a, const vector<double> &b, const vector<double> &c) {
  m = b.size(), n = c.size();
  d = vector<vector<double>>(m + 2, vector<double>(n + 2));
  for (int i = 0; i < m; ++i) {
    for (int j = 0; j < n; ++j) d[i][j] = a[i][j];
  }
  p.resize(m), q.resize(n + 1);
  for (int i = 0; i < m; ++i) p[i] = n + i, d[i][n] = -1, d[i][n + 1] = b[i];
  for (int i = 0; i < n; ++i) q[i] = i, d[m][i] = -c[i];
  q[n] = -1, d[m + 1][n] = 1;
  int r = 0;
  for (int i = 1; i < m; ++i) if (d[i][n + 1] < d[r][n + 1]) r = i;
  if (d[r][n + 1] < -eps) {
    pivot(r, n);
    if (!phase(1) || d[m + 1][n + 1] < -eps) return vector<double>(n, -inf);
    for (int i = 0; i < m; ++i) if (p[i] == -1) {
      int s = min_element(d[i].begin(), d[i].end() - 1) - d[i].begin();
      pivot(i, s);
    }
  }
  if (!phase(0)) return vector<double>(n, inf);
  vector<double> x(n);
  for (int i = 0; i < m; ++i) if (p[i] < n) x[p[i]] = d[i][n + 1];
  return x;
}
```

6.8 Subset Convolution

Description: $h(s) = \sum_{s' \subseteq s} f(s')g(s \setminus s')$

```
vector<int> SubsetConv(int n, const vector<int> &f, const vector<int> &g) {
  const int m = 1 << n;
  vector<vector<int>> a(n + 1, vector<int>(m)), b(n + 1, vector<int>(m));
  for (int i = 0; i < m; ++i) {
    a[__builtin_popcount(i)][i] = f[i];
    b[__builtin_popcount(i)][i] = g[i];
  }
  for (int i = 0; i <= n; ++i) {
    for (int j = 0; j < n; ++j) {
      for (int s = 0; s < m; ++s) {
        if (s >> j & 1) {
          a[i][s] += a[i][s ^ (1 << j)];
          b[i][s] += b[i][s ^ (1 << j)];
        }
      }
    }
  }
  vector<vector<int>> c(n + 1, vector<int>(m));
  for (int s = 0; s < m; ++s) {
    for (int i = 0; i <= n; ++i) {
      for (int j = 0; j <= i; ++j) c[i][s] += a[j][s] * b[i - j][s];
    }
  }
  for (int i = 0; i <= n; ++i) {
    for (int j = 0; j < n; ++j) {
      for (int s = 0; s < m; ++s) {
        if (s >> j & 1) c[i][s] -= c[i][s ^ (1 << j)];
      }
    }
  }
  vector<int> res(m);
  for (int i = 0; i < m; ++i) res[i] = c[__builtin_popcount(i)][i];
  return res;
}
```

6.8.1 Construction

Standard form: maximize $\mathbf{c}^T\mathbf{x}$ subject to $A\mathbf{x} \le \mathbf{b}$ and $\mathbf{x} \ge 0$.
Dual LP: minimize $\mathbf{b}^T\mathbf{y}$ subject to $A^T\mathbf{y} \ge \mathbf{c}$ and $\mathbf{y} \ge 0$.
$\bar{\mathbf{x}}$ and $\bar{\mathbf{y}}$ are optimal if and only if for all $i \in [1, n]$, either $\bar{x}_i = 0$ or $\sum_{j=1}^{m} A_{ji}\bar{y}_j = c_i$ holds and for all $i \in [1, m]$ either $\bar{y}_i = 0$ or $\sum_{j=1}^{n} A_{ij}\bar{x}_j = b_j$ holds.

1. In case of minimization, let $c'_i = -c_i$
2. $\sum_{1\le i\le n} A_{ji}x_i \ge b_j \rightarrow \sum_{1\le i\le n} -A_{ji}x_i \le -b_j$
3. $\sum_{1\le i\le n} A_{ji}x_i = b_j$
   - $\sum_{1\le i\le n} A_{ji}x_i \le b_j$
   - $\sum_{1\le i\le n} A_{ji}x_i \ge b_j$
4. If x_i has no lower bound, replace x_i with $x_i - x'_i$

6.9 Schreier–Sims Algorithm

```
namespace schreier {
int n;
vector<vector<vector<int>>> bkts, binv;
vector<vector<int>> lk;
vector<int> operator*(const vector<int> &a, const vector<int> &
    b) {
  vector<int> res(a.size());
  for (int i = 0; i < (int)a.size(); ++i) res[i] = b[a[i]];
  return res;
}
vector<int> inv(const vector<int> &a) {
  vector<int> res(a.size());
  for (int i = 0; i < (int)a.size(); ++i) res[a[i]] = i;
  return res;
}
int filter(const vector<int> &g, bool add = true) {
  n = (int)bkts.size();
  vector<int> p = g;
  for (int i = 0; i < n; ++i) {
    assert(p[i] >= 0 && p[i] < (int)lk[i].size());
    int res = lk[i][p[i]];
    if (res == -1) {
      if (add) {
        bkts[i].push_back(p);
        binv[i].push_back(inv(p));
        lk[i][p[i]] = (int)bkts[i].size() - 1;
      }
      return i;
    }
    p = p * binv[i][res];
  }
  return -1;
}
bool inside(const vector<int> &g) { return filter(g, false) ==
    -1; }
void solve(const vector<vector<int>> &gen, int _n) {
  n = _n;
  bkts.clear(), bkts.resize(n);
  binv.clear(), binv.resize(n);
  lk.clear(), lk.resize(n);
  vector<int> iden(n);
  iota(iden.begin(), iden.end(), 0);
  for (int i = 0; i < n; ++i) {
    lk[i].resize(n, -1);
    bkts[i].push_back(iden);
    binv[i].push_back(iden);
    lk[i][i] = 0;
  }
  for (int i = 0; i < (int)gen.size(); ++i) filter(gen[i]);
  queue<pair<pair<int, int>, pair<int, int>>> upd;
  for (int i = 0; i < n; ++i) {
    for (int j = i; j < n; ++j) {
      for (int k = 0; k < (int)bkts[i].size(); ++k) {
        for (int l = 0; l < (int)bkts[j].size(); ++l)
          upd.emplace(make_pair(i, k), make_pair(j, l));
      }
    }
  }
  while (!upd.empty()) {
    auto a = upd.front().first;
    auto b = upd.front().second;
    upd.pop();
    int res = filter(bkts[a.first][a.second] * bkts[b.first][b.
        second]);
    if (res == -1) continue;
    pair<int, int> pr = make_pair(res, (int)bkts[res].size() -
        1);
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < (int)bkts[i].size(); ++j) {
        if (i <= res) upd.emplace(make_pair(i, j), pr);
        if (res <= i) upd.emplace(pr, make_pair(i, j));
      }
    }
  }
}
long long size() {
  long long res = 1;
  for (int i = 0; i < n; ++i) res = res * bkts[i].size();
  return res;
}}
```

6.10 Berlekamp-Massey Algorithm

```
template <int P>
vector<int> BerlekampMassey(vector<int> x) {
  vector<int> cur, ls;
  int lf = 0, ld = 0;
  for (int i = 0; i < (int)x.size(); ++i) {
    int t = 0;
    for (int j = 0; j < (int)cur.size(); ++j)
      (t += 1LL * cur[j] * x[i - j - 1] % P) %= P;
    if (t == x[i]) continue;
    if (cur.empty()) {
      cur.resize(i + 1);
      lf = i, ld = (t + P - x[i]) % P;
      continue;
    }
    int k = 1LL * fpow(ld, P - 2, P) * (t + P - x[i]) % P;
    vector<int> c(i - lf - 1);
    c.push_back(k);
    for (int j = 0; j < (int)ls.size(); ++j)
      c.push_back(1LL * k * (P - ls[j]) % P);
    if (c.size() < cur.size()) c.resize(cur.size());
    for (int j = 0; j < (int)cur.size(); ++j)
      c[j] = (c[j] + cur[j]) % P;
    if (i - lf + (int)ls.size() >= (int)cur.size()) {
      ls = cur, lf = i;
      ld = (t + P - x[i]) % P;
    }
    cur = c;
  }
  return cur;
}
```

6.11 Fast Linear Recurrence

```
template <int P>
int LinearRec(const vector<int> &s, const vector<int> &coeff,
    int k) {
    int n = s.size();
    auto Combine = [&](const auto &a, const auto &b) {
        vector<int> res(n * 2 + 1);
        for (int i = 0; i <= n; ++i) {
            for (int j = 0; j <= n; ++j)
                (res[i + j] += 1LL * a[i] * b[j] % P) %= P;
        }
        for (int i = 2 * n; i > n; --i) {
            for (int j = 0; j < n; ++j)
                (res[i - 1 - j] += 1LL * res[i] * coeff[j] % P)
                    %= P;
        }
        res.resize(n + 1);
        return res;
    };
    vector<int> p(n + 1), e(n + 1);
    p[0] = e[1] = 1;
    for (; k > 0; k >>= 1) {
        if (k & 1) p = Combine(p, e);
        e = Combine(e, e);
    }
    int res = 0;
    for (int i = 0; i < n; ++i) (res += 1LL * p[i + 1] * s[i] %
         P) %= P;
    return res;
}
```

6.12 Prime check and factorize

```
i64 mul(i64 a, i64 b, i64 mod) {}
i64 qpow(i64 x, i64 p, i64 mod) {}
bool isPrime(i64 n) {
    if (n == 1) { return false; }
    int r = __builtin_ctzll(n - 1);
    i64 d = n - 1 >> r;
    auto checkComposite = [&](i64 p) {
        i64 x = qpow(p, d, n);
```

```
        if (x == 1 || x == n - 1) { return false; }
        for (int i = 1; i < r; i++) {
            x = mul(x, x, n);
            if (x == n - 1) { return false; }
        }
        return true;
    };
    for (auto p : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37})
        {
        if (n == p) {
            return true;
        } else if (checkComposite(p)) {
            return false;
        }
    }
    return true;
}
vector<i64> pollardRho(i64 n) {
    vector<i64> res;
    auto work = [&](auto work, i64 n) {
        if (n <= 10000) {
            for (int i = 2; i * i <= n; i++) {
                while (n % i == 0) {
                    res.push_back(i);
                    n /= i;
                }
            }
            if (n > 1) { res.push_back(n); }
            return;
        } else if (isPrime(n)) {
            res.push_back(n);
            return;
        }
        i64 x0 = 2;
        auto f = [&](i64 x) { return (mul(x, x, n) + 1) % n; };
        while (true) {
            i64 x = x0, y = x0, d = 1, power = 1, lam = 0, v =
                1;
            while (d == 1) {
                y = f(y);
                ++lam;
                v = mul(v, abs(x - y), n);
                if (lam % 127 == 0) {
                    d = gcd(v, n);
                    v = 1;
                }
                if (power == lam) {
                    x = y;
                    power *= 2;
                    lam = 0;
                    d = gcd(v, n);
                    v = 1;
                }
            }
            if (d != n) {
                work(work, d);
                work(work, n / d);
                return;
            }
            ++x0;
        }
    };
    work(work, n);
    sort(res.begin(), res.end());
    return res;
}
```

6.13 Meissel–Lehmer Algorithm

```
int64_t PrimeCount(int64_t n) {
  if (n <= 1) return 0;
  const int v = sqrt(n);
  vector<int> smalls(v + 1);
  for (int i = 2; i <= v; ++i) smalls[i] = (i + 1) / 2;
  int s = (v + 1) / 2;
  vector<int> roughs(s);
  for (int i = 0; i < s; ++i) roughs[i] = 2 * i + 1;
  vector<int64_t> larges(s);
  for (int i = 0; i < s; ++i) larges[i] = (n / (2 * i + 1) + 1)
      / 2;
  vector<bool> skip(v + 1);
  int pc = 0;
  for (int p = 3; p <= v; ++p) {
    if (smalls[p] > smalls[p - 1]) {
      int q = p * p;
      pc++;
      if (1LL * q * q > n) break;
      skip[p] = true;
      for (int i = q; i <= v; i += 2 * p) skip[i] = true;
      int ns = 0;
      for (int k = 0; k < s; ++k) {
        int i = roughs[k];
        if (skip[i]) continue;
        int64_t d = 1LL * i * p;
        larges[ns] = larges[k] - (d <= v ? larges[smalls[d] -
            pc] : smalls[n / d]) + pc;
        roughs[ns++] = i;
      }
      s = ns;
      for (int j = v / p; j >= p; --j) {
        int c = smalls[j] - pc;
        for (int i = j * p, e = min(i + p, v + 1); i < e; ++i)
            smalls[i] -= c;
      }
    }
  }
  for (int k = 1; k < s; ++k) {
    const int64_t m = n / roughs[k];
    int64_t s = larges[k] - (pc + k - 1);
    for (int l = 1; l < k; ++l) {
      int p = roughs[l];
      if (1LL * p * p > m) break;
      s -= smalls[m / p] - (pc + l - 1);
    }
    larges[0] -= s;
  }
  return larges[0];
}
```

6.14 Discrete Logarithm

```
// return min x >= 0 s.t. a ^ x = b mod m, 0 ^ 0 = 1, -1 if no
    solution
// (I think) if you want x > 0 (m != 1), remove if (b == k)
    return add;
int discreteLog(int a, int b, int m) {
    if (m == 1) {
        return 0;
    }
    a %= m, b %= m;
    int k = 1, add = 0, g;
    while ((g = gcd(a, m)) > 1) {
        if (b == k) {
            return add;
        } else if (b % g) {
            return -1;
        }
        b /= g, m /= g, ++add;
        k = 1LL * k * a / g % m;
    }
    if (b == k) {
        return add;
    }
    int n = sqrt(m) + 1;
    int an = 1;
    for (int i = 0; i < n; ++i) {
        an = 1LL * an * a % m;
    }
    unordered_map<int, int> vals;
    for (int q = 0, cur = b; q < n; ++q) {
        vals[cur] = q;
        cur = 1LL * a * cur % m;
    }
    for (int p = 1, cur = k; p <= n; ++p) {
        cur = 1LL * cur * an % m;
        if (vals.count(cur)) {
            int ans = n * p - vals[cur] + add;
            return ans;
        }
    }
    return -1;
}
```

6.15 Quadratic Residue

```
// rng
int jacobi(int a, int m) {
    int s = 1;
    while (m > 1) {
        a %= m;
        if (a == 0) { return 0; }
        int r = __builtin_ctz(a);
        if (r % 2 == 1 && (m + 2 & 4) != 0) { s = -s; }
        a >>= r;
        if ((a & m & 2) != 0) { s = -s; }
        swap(a, m);
```

```
    }
    return s;
}
int quadraticResidue(int a, int p) {
    if (p == 2) { return a % 2; }
    int j = jacobi(a, p);
    if (j == 0 || j == -1) { return j; }
    int b, d;
    while (true) {
        b = rng() % p;
        d = (1LL * b * b + p - a) % p;
        if (jacobi(d, p) == -1) { break; }
    }
    int f0 = b, f1 = 1, g0 = 1, g1 = 0, tmp;
    for (int e = p + 1 >> 1; e > 0; e >>= 1) {
        if (e % 2 == 1) {
            tmp = (1LL * g0 * f0 + 1LL * d * g1 % p * f1 % p) %
                 p;
            g1 = (1LL * g0 * f1 + 1LL * g1 * f0) % p;
            g0 = tmp;
        }
        tmp = (1LL * f0 * f0 + 1LL * d * f1 % p * f1 % p) % p;
        f1 = 2LL * f0 * f1 % p;
        f0 = tmp;
    }
    return g0;
}
```

6.16 Characteristic Polynomial

```
vector<vector<int>> Hessenberg(const vector<vector<int>> &A) {
  int N = A.size();
  vector<vector<int>> H = A;
  for (int i = 0; i < N - 2; ++i) {
    if (!H[i + 1][i]) {
      for (int j = i + 2; j < N; ++j) {
        if (H[j][i]) {
          for (int k = i; k < N; ++k) swap(H[i + 1][k], H[j][k
               ]);
          for (int k = 0; k < N; ++k) swap(H[k][i + 1], H[k][j
               ]);
          break;
        }
      }
    }
    if (!H[i + 1][i]) continue;
    int val = fpow(H[i + 1][i], kP - 2);
    for (int j = i + 2; j < N; ++j) {
      int coef = 1LL * val * H[j][i] % kP;
      for (int k = i; k < N; ++k) H[j][k] = (H[j][k] + 1LL * H[
           i + 1][k] * (kP - coef)) % kP;
      for (int k = 0; k < N; ++k) H[k][i + 1] = (H[k][i + 1] +
           1LL * H[k][j] * coef) % kP;
    }
  }
  return H;
}
vector<int> CharacteristicPoly(const vector<vector<int>> &A) {
  int N = A.size();
  auto H = Hessenberg(A);
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) H[i][j] = kP - H[i][j];
  }
  vector<vector<int>> P(N + 1, vector<int>(N + 1));
  P[0][0] = 1;
  for (int i = 1; i <= N; ++i) {
    P[i][0] = 0;
    for (int j = 1; j <= i; ++j) P[i][j] = P[i - 1][j - 1];
    int val = 1;
    for (int j = i - 1; j >= 0; --j) {
      int coef = 1LL * val * H[j][i - 1] % kP;
      for (int k = 0; k <= j; ++k) P[i][k] = (P[i][k] + 1LL * P
           [j][k] * coef) % kP;
      if (j) val = 1LL * val * (kP - H[j][j - 1]) % kP;
    }
  }
  if (N & 1) {
    for (int i = 0; i <= N; ++i) P[N][i] = kP - P[N][i];
  }
  return P[N];
}
```

6.17 Linear Sieve Related

```
vector<int> minp(N + 1), primes, mobius(N + 1);
mobius[1] = 1;
for (int i = 2; i <= N; i++) {
    if (!minp[i]) {
        primes.push_back(i);
        minp[i] = i;
        mobius[i] = -1;
    }
    for (int p : primes) {
        if (p > N / i) {
            break;
        }
        minp[p * i] = p;
        mobius[p * i] = -mobius[i];
        if (i % p == 0) {
            mobius[p * i] = 0;
            break;
        }
    }
}
```

6.18 Partition Function

```
void Build(int n) {
  // P[i] = the number of ways to represent i as the sum of a
       non-decreasing sequence.
  vector<pair<int, int>> g = {{0, 0}};
  for (int i = 1; g.back().second <= n; ++i) {
    g.emplace_back(i % 2 ? 1 : kMod - 1, i * (3 * i - 1) / 2);
    g.emplace_back(i % 2 ? 1 : kMod - 1, i * (3 * i + 1) / 2);
  }
  P[0] = P[1] = 1;
  for (int i = 2; i <= n; ++i) {
    for (auto it : g) {
      if (i < it.second) continue;
      (P[i] += 1LL * P[i - it.second] * it.first % kMod) %=
           kMod;
    }
  }
}
```

6.19 De Bruijn Sequence

```
int res[kN], aux[kN], a[kN], sz;
void Rec(int t, int p, int n, int k) {
  if (t > n) {
    if (n % p == 0)
      for (int i = 1; i <= p; ++i) res[sz++] = aux[i];
  } else {
    aux[t] = aux[t - p];
    Rec(t + 1, p, n, k);
    for (aux[t] = aux[t - p] + 1; aux[t] < k; ++aux[t]) Rec(t +
         1, t, n, k);
  }
}
int DeBruijn(int k, int n) {
  // return cyclic string of length k^n such that every string
       of length n using k character appears as a substring.
  if (k == 1) return res[0] = 0, 1;
  fill(aux, aux + k * n, 0);
  return sz = 0, Rec(1, 1, n, k), sz;
}
```

6.20 Floor Sum

```
// \sum {i = 0} {n} floor((a * i + b) / c)
i64 floorSum(i64 a, i64 b, i64 c, i64 n) {
    if (n < 0) { return 0; }
    if (n == 0) { return b / c; }
    if (a == 0) { return b / c * (n + 1); }
    i64 res = 0;
    if (a >= c) { res += a / c * n * (n + 1) / 2, a %= c; }
    if (b >= c) { res += b / c * (n + 1), b %= c; }
    i64 m = (a * n + b) / c;
    return res + n * m - (m == 0 ? 0 : floorSum(c, c - b - 1, a
         , m - 1));
}
```

6.21 More Floor Sum

- $m = \lfloor \frac{an+b}{c} \rfloor$

$$g(a, b, c, n) = \sum_{i=0}^{n} i \lfloor \frac{ai + b}{c} \rfloor$$

$$= \begin{cases} \lfloor \frac{a}{c} \rfloor \cdot \frac{n(n+1)(2n+1)}{6} + \lfloor \frac{b}{c} \rfloor \cdot \frac{n(n+1)}{2} & \\ +g(a \bmod c, b \bmod c, c, n), & a \geq c \vee b \geq c \\ 0, & n < 0 \vee a = 0 \\ \frac{1}{2} \cdot (n(n+1)m - f(c, c-b-1, a, m-1) & \\ -h(c, c-b-1, a, m-1)), & \text{otherwise} \end{cases}$$

$$h(a,b,c,n)=\sum_{i=0}^{n}\lfloor\frac{ai+b}{c}\rfloor^2$$

$$=\begin{cases}\lfloor\frac{a}{c}\rfloor^2\cdot\frac{n(n+1)(2n+1)}{6}+\lfloor\frac{b}{c}\rfloor^2\cdot(n+1) \\ +\lfloor\frac{a}{c}\rfloor\cdot\lfloor\frac{b}{c}\rfloor\cdot n(n+1) \\ +h(a \bmod c, b \bmod c, c, n) \\ +2\lfloor\frac{a}{c}\rfloor\cdot g(a \bmod c, b \bmod c, c, n) \\ +2\lfloor\frac{b}{c}\rfloor\cdot f(a \bmod c, b \bmod c, c, n), & a\ge c\vee b\ge c \\ 0, & n<0\vee a=0 \\ nm(m+1)-2g(c,c-b-1,a,m-1) \\ -2f(c,c-b-1,a,m-1)-f(a,b,c,n), & \text{otherwise}\end{cases}$$

6.22 Theorem

6.22.1 Kirchhoff's Theorem

Denote L be a $n\times n$ matrix as the Laplacian matrix of graph G, where $L_{ii}=d(i)$, $L_{ij}=-c$ where c is the number of edge (i,j) in G.

- The number of undirected spanning in G is $|\det(\tilde{L}_{11})|$.
- The number of directed spanning tree rooted at r in G is $|\det(\tilde{L}_{rr})|$.

6.22.2 Tutte's Matrix

Let D be a $n\times n$ matrix, where $d_{ij}=x_{ij}$ (x_{ij} is chosen uniformly at random) if $i<j$ and $(i,j)\in E$, otherwise $d_{ij}=-d_{ji}$. $\frac{rank(D)}{2}$ is the maximum matching on G.

6.22.3 Cayley's Formula

- Given a degree sequence $d_1,d_2,\ldots,d_n$ for each *labeled* vertices, there are $\frac{(n-2)!}{(d_1-1)!(d_2-1)!\cdots(d_n-1)!}$ spanning trees.
- Let $T_{n,k}$ be the number of *labeled* forests on n vertices with k components, such that vertex $1,2,\ldots,k$ belong to different components. Then $T_{n,k}=kn^{n-k-1}$.

6.22.4 Erdős–Gallai Theorem

A sequence of non-negative integers $d_1\ge d_2\ge\ldots\ge d_n$ can be represented as the degree sequence of a finite simple graph on n vertices if and only if $d_1+d_2+\ldots+d_n$ is even and

$$\sum_{i=1}^{k}d_i\le k(k-1)+\sum_{i=k+1}^{n}\min(d_i,k)$$

holds for all $1\le k\le n$.

7 Dynamic Programming

7.1 Dynamic Convex Hull

```
struct Line {
    // kx + b
  mutable i64 k, b, p;
  bool operator<(const Line& o) const { return k < o.k; }
  bool operator<(i64 x) const { return p < x; }
};
struct DynamicConvexHullMax : multiset<Line, less<>> {
  // (for doubles, use INF = 1/.0, div(a,b) = a/b)
  static constexpr i64 INF = numeric_limits<i64>::max();
  i64 div(i64 a, i64 b) {
      // floor
    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->b > y->b ? INF : -INF;
    else x->p = div(y->b - x->b, x->k - y->k);
    return x->p >= y->p;
  }
  void add(i64 k, i64 b) {
    auto z = insert({k, b, 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));
  }
  i64 query(i64 x) {
      if (empty()) {
          return -INF;
      }
    auto l = *lower_bound(x);
    return l.k * x + l.b;
  }
};
```

7.2 $1D/1D$ Convex Optimization

```
struct segment {
  int i, l, r;
  segment(int a, int b, int c): i(a), l(b), r(c) {}
};
inline long long f(int l, int r) { return dp[l] + w(l + 1, r);
    }
void solve() {
  dp[0] = 0ll;
  deque&lt;segment&gt; deq; deq.push_back(segment(0, 1, n));
  for (int i = 1; i <= n; ++i) {
    dp[i] = f(deq.front().i, i);
    while (deq.size() && deq.front().r < i + 1) deq.pop_front()
        ;
    deq.front().l = i + 1;
    segment seg = segment(i, i + 1, n);
    while (deq.size() && f(i, deq.back().l) < f(deq.back().i,
        deq.back().l)) deq.pop_back();
    if (deq.size()) {
      int d = 1048576, c = deq.back().l;
      while (d >>= 1) if (c + d <= deq.back().r) {
        if (f(i, c + d) > f(deq.back().i, c + d)) c += d;
      }
      deq.back().r = c; seg.l = c + 1;
    }
    if (seg.l <= n) deq.push_back(seg);
  }
}
```

7.3 Conditon

7.3.1 Totally Monotone (Concave/Convex)

$\forall i<i', j<j',\ B[i][j]\le B[i'][j] \implies B[i][j']\le B[i'][j']$
$\forall i<i', j<j',\ B[i][j]\ge B[i'][j] \implies B[i][j']\ge B[i'][j']$

7.3.2 Monge Condition (Concave/Convex)

$\forall i<i', j<j',\ B[i][j]+B[i'][j']\ge B[i][j']+B[i'][j]$
$\forall i<i', j<j',\ B[i][j]+B[i'][j']\le B[i][j']+B[i'][j]$

7.3.3 Optimal Split Point

If

$$B[i][j]+B[i+1][j+1]\ge B[i][j+1]+B[i+1][j]$$

then

$$H_{i,j-1}\le H_{i,j}\le H_{i+1,j}$$

8 Geometry

8.1 Basic

```
using Real = double; // modify these if needed
constexpr Real eps = 1e-9;
int sign(T x) { return (x > 0) - (x < 0); }
int sign(Real x) { return (x > eps) - (x < -eps); }
int cmp(T a, T b) { return sign(a - b); }
struct P {
    T x = 0, y = 0;
    P(T x = 0, T y = 0) : x(x), y(y) {}
    -, +*/, ==!=<, - (unary)
};
struct L {
    P<T> a, b;
    L(P<T> a = {}, P<T> b = {}) : a(a), b(b) {}
};
T dot(P<T> a, P<T> b) { return a.x * b.x + a.y * b.y; }
T square(P<T> a) { return dot(a, a); }
Real length(P<T> a) { return sqrtl(square(a)); }
Real dist(P<T> a, P<T> b) { return length(a - b); }
T cross(P<T> a, P<T> b) { return a.x * b.y - a.y * b.x; }
T cross(P<T> p, P<T> a, P<T> b) { return cross(a - p, b - p); }
P<Real> normal(P<T> a) {
    Real len = length(a);
    return P<Real>(a.x / len, a.y / len);
}
bool up(P<T> a) { return sign(a.y) > 0 || sign(a.y) == 0 &&
     sign(a.x) > 0; }
// 3 colinear? please remember to remove (0, 0)
bool polar(P<T> a, P<T> b) {
    bool ua = up(a), ub = up(b);
    return ua != ub ? ua : sign(cross(a, b)) == 1;
}
// 1/0/1 if on a->b's left/ /right
```

```cpp
int side(P<T> p, P<T> a, P<T> b) { return sign(cross(p, a, b));
    }
int side(P<T> p, L<T> l) { return side(p, l.a, l.b); }
P<T> rotate90(P<T> p) { return {-p.y, p.x}; }
P<Real> rotate(P<Real> p, Real ang) { return {p.x * cos(ang) -
     p.y * sin(ang), p.x * sin(ang) + p.y * cos(ang)}; }
Real angle(P<T> p) { return atan2(p.y, p.x); }
P<T> direction(L<T> l) { return l.b - l.a; }
bool parallel(L<T> l1, L<T> l2) { return sign(cross(direction(
     l1), direction(l2))) == 0; }
bool sameDirection(L<T> l1, L<T> l2) { return parallel(l1, l2)
     && sign(dot(direction(l1), direction(l2))) == 1; }
P<Real> projection(P<Real> p, L<Real> l) {
    auto d = direction(l);
    return l.a + d * (dot(p - l.a, d) / square(d));
}
P<Real> reflection(P<Real> p, L<Real> l) { return projection(p,
     l) * 2 - p; }
Real pointToLineDist(P<Real> p, L<Real> l) { return dist(p,
     projection(p, l)); }
// better use integers if you don't need exact coordinate
// l <= r is not explicitly required
P<Real> lineIntersection(L<T> l1, L<T> l2) { return l1.a -
     direction(l1) * (Real(cross(direction(l2), l1.a - l2.a)) /
      cross(direction(l2), direction(l1))); }
bool between(T m, T l, T r) { return cmp(l, m) == 0 || cmp(m, r
     ) == 0 || l < m != r < m; }
bool pointOnSeg(P<T> p, L<T> l) { return side(p, l) == 0 &&
     between(p.x, l.a.x, l.b.x) && between(p.y, l.a.y, l.b.y);
     }
bool pointStrictlyOnSeg(P<T> p, L<T> l) { return side(p, l) ==
     0 && sign(dot(p - l.a, direction(l))) * sign(dot(p - l.b,
     direction(l))) < 0; }
bool overlap(T l1, T r1, T l2, T r2) {
    if (l1 > r1) { swap(l1, r1); }
    if (l2 > r2) { swap(l2, r2); }
    return cmp(r1, l2) != -1 && cmp(r2, l1) != -1;
}
bool segIntersect(L<T> l1, L<T> l2) {
    auto [p1, p2] = l1;
    auto [q1, q2] = l2;
    return overlap(p1.x, p2.x, q1.x, q2.x) && overlap(p1.y, p2.
         y, q1.y, q2.y) &&
            side(p1, l2) * side(p2, l2) <= 0 &&
            side(q1, l1) * side(q2, l1) <= 0;
}
// parallel intersecting is false
bool segStrictlyIntersect(L<T> l1, L<T> l2) {
    auto [p1, p2] = l1;
    auto [q1, q2] = l2;
    return side(p1, l2) * side(p2, l2) < 0 &&
           side(q1, l1) * side(q2, l1) < 0;
}
// parallel or intersect at source doesn't count
bool rayIntersect(L<T> l1, L<T> l2) {
    int x = sign(cross(l1.b - l1.a, l2.b - l2.a));
    return x == 0 ? false : side(l1.a, l2) == x && side(l2.a,
         l1) == -x;
}
Real pointToSegDist(P<T> p, L<T> l) {
    P<Real> q = projection(p, l);
    if (pointOnSeg(q, l)) {
        return dist(p, q);
    } else {
        return min(dist(p, l.a), dist(p, l.b));
    }
}
Real segDist(L<T> l1, L<T> l2) {
    if (segIntersect(l1, l2)) { return 0; }
  return min({pointToSegDist(l1.a, l2), pointToSegDist(l1.b, l2
      ),
            pointToSegDist(l2.a, l1), pointToSegDist(l2.b, l1)
                 });
}
// 2 times area
T area(vector<P<T>> a) {
    T res = 0;
    int n = a.size();
    for (int i = 0; i < n; i++) { res += cross(a[i], a[(i + 1)
         % n]); }
    return res;
}
bool pointInPoly(P<T> p, vector<P<T>> a) {
    int n = a.size(), res = 0;
    for (int i = 0; i < n; i++) {
        P<T> u = a[i], v = a[(i + 1) % n];
        if (pointOnSeg(p, {u, v})) { 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
             ; }
        res ^= cross(p, u, v) > 0;
    }
    return res;
}
```

8.2 Convex Hull and related

```cpp
vector<P<T>> convexHull(vector<P<T>> a) {
    int n = a.size();
    if (n <= 1) { return a; }
    sort(a.begin(), a.end());
    vector<P<T>> b(2 * n);
    int j = 0;
    for (int i = 0; i < n; b[j++] = a[i++]) {
        while (j >= 2 && side(b[j - 2], b[j - 1], a[i]) <= 0) {
             j--; }
    }
    for (int i = n - 2, k = j; i >= 0; b[j++] = a[i--]) {
        while (j > k && side(b[j - 2], b[j - 1], a[i]) <= 0) {
            j--; }
    }
    b.resize(j - 1);
    return b;
}
// warning : if all point on same line will return {1, 2, 3, 2}
vector<P<T>> convexHullNonStrict(vector<P<T>> a) {
    sort(a.begin(), a.end());
    a.erase(unique(a.begin(), a.end()), a.end());
    int n = a.size();
    if (n == 1) { return a; }
    vector<P<T>> b(2 * n);
    int j = 0;
    for (int i = 0; i < n; b[j++] = a[i++]) {
        while (j >= 2 && side(b[j - 2], b[j - 1], a[i]) < 0) {
            j--; }
    }
    for (int i = n - 2, k = j; i >= 0; b[j++] = a[i--]) {
        while (j > k && side(b[j - 2], b[j - 1], a[i]) < 0) { j
            --; }
    }
    b.resize(j - 1);
    return b;
}
```

8.3 Half Plane Intersection

```cpp
vector<P<Real>> halfPlaneIntersection(vector<L<Real>> a) {
    sort(a.begin(), a.end(), [&](auto l1, auto l2) {
        if (sameDirection(l1, l2)) {
            return side(l1.a, l2) > 0;
        } else {
            return polar(direction(l1), direction(l2));
        }
    });
    deque<L<Real>> dq;
    auto check = [&](L<Real> l, L<Real> l1, L<Real> l2) {
         return side(lineIntersection(l1, l2), l) > 0; };
    for (int i = 0; i < int(a.size()); i++) {
        if (i > 0 && sameDirection(a[i], a[i - 1])) { continue;
             }
        while (int(dq.size()) > 1 && !check(a[i], dq.end()[-2],
             dq.back())) { dq.pop_back(); }
        while (int(dq.size()) > 1 && !check(a[i], dq[1], dq[0])
            ) { dq.pop_front(); }
        dq.push_back(a[i]);
    }
    while (int(dq.size()) > 2 && !check(dq[0], dq.end()[-2], dq
         .back())) { dq.pop_back(); }
    while (int(dq.size()) > 2 && !check(dq.back(), dq[1], dq
         [0])) { dq.pop_front(); }
    vector<P<Real>> res;
    dq.push_back(dq[0]);
    for (int i = 0; i + 1 < int(dq.size()); i++) { res.
         push_back(lineIntersection(dq[i], dq[i + 1])); }
    return res;
}
```

8.4 Triangle Centers

```cpp
// radius: (a + b + c) * r / 2 = A or pointToLineDist
P<Real> inCenter(P<Real> a, P<Real> b, P<Real> c) {
    Real la = length(b - c), lb = length(c - a), lc = length(a
         - b);
```

```
    return (a * la + b * lb + c * lc) / (la + lb + lc);
}
// used in min enclosing circle
P<Real> circumCenter(P<Real> a, P<Real> b, P<Real> c) {
    P<Real> ba = b - a, ca = c - a;
    Real db = square(ba), dc = square(ca), d = 2 * cross(ba, ca
        );
    return a - P<Real>(ba.y * dc - ca.y * db, ca.x * db - ba.x
        * dc) / d;
}
P<Real> orthoCenter(P<Real> a, P<Real> b, P<Real> c) {
    L<Real> u(c, P<Real>(c.x - a.y + b.y, c.y + a.x - b.x));
    L<Real> v(b, P<Real>(b.x - a.y + c.y, b.y + a.x - c.x));
    return lineIntersection(u, v);
}
```

8.5 Circle

```
const Real PI = acos(-1);
struct Circle {
    P<Real> o;
    Real r;
    Circle(P<Real> o = {}, Real r = 0) : o(o), r(r) {}
};
// actually counts number of tangent lines
int typeOfCircles(Circle c1, Circle c2) {
    auto [o1, r1] = c1;
    auto [o2, r2] = c2;
    Real d = dist(o1, o2);
    if (cmp(d, r1 + r2) == 1) { return 4; }
    if (cmp(d, r1 + r2) == 0) { return 3; }
    if (cmp(d, abs(r1 - r2)) == 1) { return 2; }
    if (cmp(d, abs(r1 - r2)) == 0) { return 1; }
    return 0;
}
// aligned l.a -> l.b;
vector<P<Real>> circleLineIntersection(Circle c, L<Real> l) {
    P<Real> p = projection(c.o, l);
    Real h = c.r * c.r - square(p - c.o);
    if (sign(h) < 0) { return {}; }
    P<Real> q = normal(direction(l)) * sqrtl(c.r * c.r - square
        (p - c.o));
    return {p - q, p + q};
}
// circles shouldn't be identical
// duplicated if only one intersection, aligned c1
    counterclockwise
vector<P<Real>> circleIntersection(Circle c1, Circle c2) {
    int type = typeOfCircles(c1, c2);
    if (type == 0 || type == 4) { return {}; }
    auto [o1, r1] = c1;
    auto [o2, r2] = c2;
    Real d = clamp(dist(o1, o2), abs(r1 - r2), r1 + r2);
    Real y = (r1 * r1 + d * d - r2 * r2) / (2 * d), x = sqrtl(
        r1 * r1 - y * y);
    P<Real> dir = normal(o2 - o1), q1 = o1 + dir * y, q2 =
        rotate90(dir) * x;
    return {q1 - q2, q1 + q2};
}
// counterclockwise, on circle -> no tangent
vector<P<Real>> pointCircleTangent(P<Real> p, Circle c) {
    Real x = square(p - c.o), d = x - c.r * c.r;
    if (sign(d) <= 0) { return {}; }
    P<Real> q1 = c.o + (p - c.o) * (c.r * c.r / x), q2 =
        rotate90(p - c.o) * (c.r * sqrt(d) / x);
  return {q1 - q2, q1 + q2};
}
// one-point tangent lines are not returned
vector<L<Real>> externalTangent(Circle c1, Circle c2) {
    auto [o1, r1] = c1;
    auto [o2, r2] = c2;
    vector<L<Real>> res;
    if (cmp(r1, r2) == 0) {
        P dr = rotate90(normal(o2 - o1)) * r1;
        res.emplace_back(o1 + dr, o2 + dr);
        res.emplace_back(o1 - dr, o2 - dr);
    } else {
        P p = (o2 * r1 - o1 * r2) / (r1 - r2);
        auto ps = pointCircleTangent(p, c1), qs =
            pointCircleTangent(p, c2);
        for (int i = 0; i < int(min(ps.size(), qs.size())); i
            ++) { res.emplace_back(ps[i], qs[i]); }
    }
    return res;
}
vector<L<Real>> internalTangent(Circle c1, Circle c2) {
    auto [o1, r1] = c1;
    auto [o2, r2] = c2;
    vector<L<Real>> res;
    P<Real> p = (o1 * r2 + o2 * r1) / (r1 + r2);
    auto ps = pointCircleTangent(p, c1), qs =
        pointCircleTangent(p, c2);
    for (int i = 0; i < int(min(ps.size(), qs.size())); i++) {
        res.emplace_back(ps[i], qs[i]); }
    return res;
}
// OAB and circle directed area
Real triangleCircleIntersectionArea(P<Real> p1, P<Real> p2,
    Real r) {
    auto angle = [&](P<Real> p1, P<Real> p2) { return atan2l(
        cross(p1, p2), dot(p1, p2)); };
    vector<P<Real>> v = circleLineIntersection(Circle(P<Real>()
        , r), L<Real>(p1, p2));
    if (v.empty()) { return r * r * angle(p1, p2) / 2; }
    bool b1 = cmp(square(p1), r * r) == 1, b2 = cmp(square(p2),
         r * r) == 1;
    if (b1 && b2) {
        if (sign(dot(p1 - v[0], p2 - v[0])) <= 0 && sign(dot(p1
            - v[0], p2 - v[0])) <= 0) {
            return r * r * (angle(p1, v[0]) + angle(v[1], p2))
                / 2 + cross(v[0], v[1]) / 2;
        } else {
            return r * r * angle(p1, p2) / 2;
        }
    } else if (b1) {
        return (r * r * angle(p1, v[0]) + cross(v[0], p2)) / 2;
    } else if (b2) {
        return (cross(p1, v[1]) + r * r * angle(v[1], p2)) / 2;
    } else {
        return cross(p1, p2) / 2;
    }
}
Real polyCircleIntersectionArea(const vector<P<Real>> &a,
    Circle c) {
    int n = a.size();
    Real ans = 0;
    for (int i = 0; i < n; i++) {
        ans += triangleCircleIntersectionArea(a[i], a[(i + 1) %
            n], c.r);
    }
    return ans;
}
Real circleIntersectionArea(Circle a, Circle b) {
    int t = typeOfCircles(a, b);
    if (t >= 3) {
        return 0;
    } else if (t <= 1) {
        Real r = min(a.r, b.r);
        return r * r * PI;
    }
    Real res = 0, d = dist(a.o, b.o);
    for (int i = 0; i < 2; ++i) {
        Real alpha = acos((b.r * b.r + d * d - a.r * a.r) / (2
            * b.r * d));
        Real s = alpha * b.r * b.r;
        Real t = b.r * b.r * sin(alpha) * cos(alpha);
        res += s - t;
        swap(a, b);
    }
    return res;
}
```

8.6 Closest Pair

```
double closest_pair(int l, int r) {
  // p should be sorted increasingly according to the x-
    coordinates.
  if (l == r) return 1e9;
  if (r - l == 1) return dist(p[l], p[r]);
  int m = (l + r) >> 1;
  double d = min(closest_pair(l, m), closest_pair(m + 1, r));
  vector<int> vec;
  for (int i = m; i >= l && fabs(p[m].x - p[i].x) < d; --i) vec
      .push_back(i);
  for (int i = m + 1; i <= r && fabs(p[m].x - p[i].x) < d; ++i)
      vec.push_back(i);
  sort(vec.begin(), vec.end(), [&](int a, int b) { return p[a].
      y < p[b].y; });
  for (int i = 0; i < vec.size(); ++i) {
    for (int j = i + 1; j < vec.size() && fabs(p[vec[j]].y - p[
        vec[i]].y) < d; ++j) {
      d = min(d, dist(p[vec[i]], p[vec[j]]));
    }
  }
```

```
  return d;
}
```

8.7 Area of Union of Circles

```
vector<pair<double, double>> CoverSegment(C &a, C &b) {
  double d = (a.c - b.c).abs();
  vector<pair<double, double>> res;
  if (same(a.r + b.r, d)) ;
  else if (d <= abs(a.r - b.r) + eps) {
    if (a.r < b.r) res.emplace_back(0, 2 * pi);
  } else if (d < abs(a.r + b.r) - eps) {
    double o = acos((sq(a.r) + sq(d) - sq(b.r)) / (2 * a.r * d)
        ), z = (b.c - a.c).angle();
    if (z < 0) z += 2 * pi;
    double l = z - o, r = z + o;
    if (l < 0) l += 2 * pi;
    if (r > 2 * pi) r -= 2 * pi;
    if (l > r) res.emplace_back(l, 2 * pi), res.emplace_back(0,
         r);
    else res.emplace_back(l, r);
  }
  return res;
}
double CircleUnionArea(vector<C> c) { // circle should be
    identical
  int n = c.size();
  double a = 0, w;
  for (int i = 0; w = 0, i < n; ++i) {
    vector<pair<double, double>> s = {{2 * pi, 9}}, z;
    for (int j = 0; j < n; ++j) if (i != j) {
      z = CoverSegment(c[i], c[j]);
      for (auto &e : z) s.push_back(e);
    }
    sort(s.begin(), s.end());
    auto F = [&] (double t) { return c[i].r * (c[i].r * t + c[i
        ].c.x * sin(t) - c[i].c.y * cos(t)); };
    for (auto &e : s) {
      if (e.first > w) a += F(e.first) - F(w);
      w = max(w, e.second);
    }
  }
  return a * 0.5;
}
```

8.8 3D Convex Hull

```
double absvol(const P a,const P b,const P c,const P d) {
  return abs(((b-a)^(c-a))*(d-a))/6;
}
struct convex3D {
  static const int maxn=1010;
  struct T{
    int a,b,c;
    bool res;
    T(){}
    T(int a,int b,int c,bool res=1):a(a),b(b),c(c),res(res){}
  };
  int n,m;
  P p[maxn];
  T f[maxn*8];
  int id[maxn][maxn];
  bool on(T &t,P &q){
    return ((p[t.c]-p[t.b])^(p[t.a]-p[t.b]))*(q-p[t.a])>eps;
  }
  void meow(int q,int a,int b){
    int g=id[a][b];
    if(f[g].res){
      if(on(f[g],p[q]))dfs(q,g);
      else{
        id[q][b]=id[a][q]=id[b][a]=m;
        f[m++]=T(b,a,q,1);
      }
    }
  }
  void dfs(int p,int i){
    f[i].res=0;
    meow(p,f[i].b,f[i].a);
    meow(p,f[i].c,f[i].b);
    meow(p,f[i].a,f[i].c);
  }
  void operator()(){
    if(n<4)return;
    if([&](){
        for(int i=1;i<n;++i)if(abs(p[0]-p[i])>eps)return swap(p
            [1],p[i]),0;
        return 1;
        }() || [&](){
        for(int i=2;i<n;++i)if(abs((p[0]-p[i])^(p[1]-p[i]))>eps
            )return swap(p[2],p[i]),0;
        return 1;
        }() || [&](){
        for(int i=3;i<n;++i)if(abs(((p[1]-p[0])^(p[2]-p[0]))*(p
            [i]-p[0]))>eps)return swap(p[3],p[i]),0;
        return 1;
        }())return;
    for(int i=0;i<4;++i){
      T t((i+1)%4,(i+2)%4,(i+3)%4,1);
      if(on(t,p[i]))swap(t.b,t.c);
      id[t.a][t.b]=id[t.b][t.c]=id[t.c][t.a]=m;
      f[m++]=t;
    }
    for(int i=4;i<n;++i)for(int j=0;j<m;++j)if(f[j].res && on(f
        [j],p[i])){
      dfs(i,j);
      break;
    }
    int mm=m; m=0;
    for(int i=0;i<mm;++i)if(f[i].res)f[m++]=f[i];
  }
  bool same(int i,int j){
    return !(absvol(p[f[i].a],p[f[i].b],p[f[i].c],p[f[j].a])>
        eps || absvol(p[f[i].a],p[f[i].b],p[f[i].c],p[f[j].b])
        >eps || absvol(p[f[i].a],p[f[i].b],p[f[i].c],p[f[j].c
        ])>eps);
  }
  int faces(){
    int r=0;
    for(int i=0;i<m;++i){
      int iden=1;
      for(int j=0;j<i;++j)if(same(i,j))iden=0;
      r+=iden;
    }
    return r;
  }
} tb;
```

8.9 Delaunay Triangulation

```
const P<i64> pINF = P<i64>(1e18, 1e18);
using i128 = __int128_t;
struct Quad {
  P<i64> origin;
  Quad *rot = nullptr, *onext = nullptr;
  bool used = false;
  Quad* rev() const { return rot->rot; }
  Quad* lnext() const { return rot->rev()->onext->rot; }
  Quad* oprev() const { return rot->onext->rot; }
  P<i64> dest() const { return rev()->origin; }
};
Quad* makeEdge(P<i64> from, P<i64> to) {
  Quad *e1 = new Quad, *e2 = new Quad, *e3 = new Quad, *e4 =
      new Quad;
  e1->origin = from;
  e2->origin = to;
  e3->origin = e4->origin = pINF;
  e1->rot = e3;
  e2->rot = e4;
  e3->rot = e2;
  e4->rot = e1;
  e1->onext = e1;
  e2->onext = e2;
  e3->onext = e4;
  e4->onext = e3;
  return e1;
}
void splice(Quad *a, Quad *b) {
  swap(a->onext->rot->onext, b->onext->rot->onext);
  swap(a->onext, b->onext);
}
void delEdge(Quad *e) {
  splice(e, e->oprev());
  splice(e->rev(), e->rev()->oprev());
  delete e->rev()->rot;
  delete e->rev();
  delete e->rot;
  delete e;
}
Quad *connect(Quad *a, Quad *b) {
  Quad *e = makeEdge(a->dest(), b->origin);
  splice(e, a->lnext());
  splice(e->rev(), b);
  return e;
}
bool onLeft(P<i64> p, Quad *e) { return side(p, e->origin, e->
    dest()) > 0; }
```

```
bool onRight(P<i64> p, Quad *e) { return side(p, e->origin, e->dest()) < 0; }
template <class T>
T det3(T a1, T a2, T a3, T b1, T b2, T b3, T c1, T c2, T c3) {
  return a1 * (b2 * c3 - c2 * b3) - a2 * (b1 * c3 - c1 * b3) + a3 * (b1 * c2 - c1 * b2);
}
bool inCircle(P<i64> a, P<i64> b, P<i64> c, P<i64> d) {
  auto f = [&](P<i64> a, P<i64> b, P<i64> c) {
    return det3<i128>(a.x, a.y, square(a), b.x, b.y, square(b), c.x, c.y, square(c));
  };
  i128 det = f(a, c, d) + f(a, b, c) - f(b, c, d) - f(a, b, d);
  return det > 0;
}
pair<Quad*, Quad*> build(int l, int r, vector<P<i64>> &p) {
  if (r - l == 2) {
    Quad *res = makeEdge(p[l], p[l + 1]);
    return pair(res, res->rev());
  } else if (r - l == 3) {
    Quad *a = makeEdge(p[l], p[l + 1]), *b = makeEdge(p[l + 1], p[l + 2]);
    splice(a->rev(), b);
    int sg = sign(cross(p[l], p[l + 1], p[l + 2]));
    if (sg == 0) { return pair(a, b->rev()); }
    Quad *c = connect(b, a);
    if (sg == 1) {
      return pair(a, b->rev());
    } else {
      return pair(c->rev(), c);
    }
  }
  int m = l + r >> 1;
  auto [ldo, ldi] = build(l, m, p);
  auto [rdi, rdo] = build(m, r, p);
  while (true) {
    if (onLeft(rdi->origin, ldi)) {
      ldi = ldi->lnext();
      continue;
    }
    if (onRight(ldi->origin, rdi)) {
      rdi = rdi->rev()->onext;
      continue;
    }
    break;
  }
  Quad *basel = connect(rdi->rev(), ldi);
  auto valid = [&](Quad *e) { return onRight(e->dest(), basel); };
  if (ldi->origin == ldo->origin) { ldo = basel->rev(); }
  if (rdi->origin == rdo->origin) { rdo = basel; }
  while (true) {
    Quad *lcand = basel->rev()->onext;
    if (valid(lcand)) {
      while (inCircle(basel->dest(), basel->origin, lcand->dest(), lcand->onext->dest())) {
        Quad *t = lcand->onext;
        delEdge(lcand);
        lcand = t;
      }
    }
    Quad *rcand = basel->oprev();
    if (valid(rcand)) {
      while (inCircle(basel->dest(), basel->origin, rcand->dest(), rcand->oprev()->dest())) {
        Quad *t = rcand->oprev();
        delEdge(rcand);
        rcand = t;
      }
    }
    if (!valid(lcand) && !valid(rcand)) { break; }
    if (!valid(lcand) || valid(rcand) && inCircle(lcand->dest(), lcand->origin, rcand->origin, rcand->dest())) {
      basel = connect(rcand, basel->rev());
    } else {
      basel = connect(basel->rev(), lcand->rev());
    }
  }
  return pair(ldo, rdo);
}
vector<array<P<i64>, 3>> delaunay(vector<P<i64>> p) {
  sort(p.begin(), p.end());
  auto res = build(0, p.size(), p);
  Quad *e = res.first;
  vector<Quad*> edges = {e};
  while (sign(cross(e->onext->dest(), e->dest(), e->origin)) == -1) { e = e->onext; }
  auto add = [&]() {
    Quad *cur = e;
    do {
      cur->used = true;
      p.push_back(cur->origin);
      edges.push_back(cur->rev());
      cur = cur->lnext();
    } while (cur != e);
  };
  add();
  p.clear();
  int i = 0;
  while (i < int(edges.size())) { if (!(e = edges[i++])->used) { add(); }}
  vector<array<P<i64>, 3>> ans(p.size() / 3);
  for (int i = 0; i < int(p.size()); i++) { ans[i / 3][i % 3] = p[i]; }
  return ans;
}
```

9 Miscellaneous

9.1 Cactus

```
// a component contains no articulation point, so P2 is a component
// resulting bct is rooted
struct BlockCutTree {
    int n, square = 0, cur = 0;
    vector<int> low, dfn, stk;
    vector<vector<int>> adj, bct;
    BlockCutTree(int n) : n(n), low(n), dfn(n, -1), adj(n), bct(n) {}
    void build() { dfs(0); }
    void addEdge(int u, int v) { adj[u].push_back(v), adj[v].push_back(u); }
    void dfs(int u) {
        low[u] = dfn[u] = cur++;
        stk.push_back(u);
        for (auto v : adj[u]) {
            if (dfn[v] == -1) {
                dfs(v);
                low[u] = min(low[u], low[v]);
                if (low[v] == dfn[u]) {
                    bct.emplace_back();
                    int x;
                    do {
                        x = stk.back();
                        stk.pop_back();
                        bct.back().push_back(x);
                    } while (x != v);
                    bct[u].push_back(n + square);
                    square++;
                }
            } else {
                low[u] = min(low[u], dfn[v]);
            }
        }
    }
};
```

9.2 Dancing Links

```
namespace dlx {
int lt[maxn], rg[maxn], up[maxn], dn[maxn], cl[maxn], rw[maxn], bt[maxn], s[maxn], head, sz, ans;
void init(int c) {
  for (int i = 0; i < c; ++i) {
    up[i] = dn[i] = bt[i] = i;
    lt[i] = i == 0 ? c : i - 1;
    rg[i] = i == c - 1 ? c : i + 1;
    s[i] = 0;
  }
  rg[c] = 0, lt[c] = c - 1;
  up[c] = dn[c] = -1;
  head = c, sz = c + 1;
}
void insert(int r, const vector<int> &col) {
  if (col.empty()) return;
  int f = sz;
  for (int i = 0; i < (int)col.size(); ++i) {
    int c = col[i], v = sz++;
    dn[bt[c]] = v;
    up[v] = bt[c], bt[c] = v;
    rg[v] = (i + 1 == (int)col.size() ? f : v + 1);
```

```
    rw[v] = r, cl[v] = c;
    ++s[c];
    if (i > 0) lt[v] = v - 1;
  }
  lt[f] = sz - 1;
}
void remove(int c) {
  lt[rg[c]] = lt[c], rg[lt[c]] = rg[c];
  for (int i = dn[c]; i != c; i = dn[i]) {
    for (int j = rg[i]; j != i; j = rg[j])
      up[dn[j]] = up[j], dn[up[j]] = dn[j], --s[cl[j]];
  }
}
void restore(int c) {
  for (int i = up[c]; i != c; i = up[i]) {
    for (int j = lt[i]; j != i; j = lt[j])
      ++s[cl[j]], up[dn[j]] = j, dn[up[j]] = j;
  }
  lt[rg[c]] = c, rg[lt[c]] = c;
}
// Call dlx::make after inserting all rows.
void make(int c) {
  for (int i = 0; i < c; ++i)
    dn[bt[i]] = i, up[i] = bt[i];
}
void dfs(int dep) {
  if (dep >= ans) return;
  if (rg[head] == head) return ans = dep, void();
  if (dn[rg[head]] == rg[head]) return;
  int c = rg[head];
  int w = c;
  for (int x = c; x != head; x = rg[x]) if (s[x] < s[w]) w = x;
  remove(w);
  for (int i = dn[w]; i != w; i = dn[i]) {
    for (int j = rg[i]; j != i; j = rg[j]) remove(cl[j]);
    dfs(dep + 1);
    for (int j = lt[i]; j != i; j = lt[j]) restore(cl[j]);
  }
  restore(w);
}
int solve() {
  ans = 1e9, dfs(0);
  return ans;
}}
```

9.3 Offline Dynamic MST

```
int cnt[maxn], cost[maxn], st[maxn], ed[maxn];
pair<int, int> qr[maxn];
// qr[i].first = id of edge to be changed, qr[i].second =
    weight after operation
// cnt[i] = number of operation on edge i
// call solve(0, q - 1, v, 0), where v contains edges i such
    that cnt[i] == 0

void contract(int l, int r, vector<int> v, vector<int> &x,
    vector<int> &y) {
  sort(v.begin(), v.end(), [&](int i, int j) {
      if (cost[i] == cost[j]) return i < j;
      return cost[i] < cost[j];
      });
  djs.save();
  for (int i = l; i <= r; ++i) djs.merge(st[qr[i].first], ed[qr
    [i].first]);
  for (int i = 0; i < (int)v.size(); ++i) {
    if (djs.find(st[v[i]]) != djs.find(ed[v[i]])) {
      x.push_back(v[i]);
      djs.merge(st[v[i]], ed[v[i]]);
    }
  }
  djs.undo();
  djs.save();
  for (int i = 0; i < (int)x.size(); ++i) djs.merge(st[x[i]],
    ed[x[i]]);
  for (int i = 0; i < (int)v.size(); ++i) {
    if (djs.find(st[v[i]]) != djs.find(ed[v[i]])) {
      y.push_back(v[i]);
      djs.merge(st[v[i]], ed[v[i]]);
    }
  }
  djs.undo();
}

void solve(int l, int r, vector<int> v, long long c) {
  if (l == r) {
    cost[qr[l].first] = qr[l].second;
    if (st[qr[l].first] == ed[qr[l].first]) {
      printf("%lld\n", c);
      return;
    }
    int minv = qr[l].second;
    for (int i = 0; i < (int)v.size(); ++i) minv = min(minv,
       cost[v[i]]);
    printf("%lld\n", c + minv);
    return;
  }
  int m = (l + r) >> 1;
  vector<int> lv = v, rv = v;
  vector<int> x, y;
  for (int i = m + 1; i <= r; ++i) {
    cnt[qr[i].first]--;
    if (cnt[qr[i].first] == 0) lv.push_back(qr[i].first);
  }
  contract(l, m, lv, x, y);
  long long lc = c, rc = c;
  djs.save();
  for (int i = 0; i < (int)x.size(); ++i) {
    lc += cost[x[i]];
    djs.merge(st[x[i]], ed[x[i]]);
  }
  solve(l, m, y, lc);
  djs.undo();
  x.clear(), y.clear();
  for (int i = m + 1; i <= r; ++i) cnt[qr[i].first]++;
  for (int i = l; i <= m; ++i) {
    cnt[qr[i].first]--;
    if (cnt[qr[i].first] == 0) rv.push_back(qr[i].first);
  }
  contract(m + 1, r, rv, x, y);
  djs.save();
  for (int i = 0; i < (int)x.size(); ++i) {
    rc += cost[x[i]];
    djs.merge(st[x[i]], ed[x[i]]);
  }
  solve(m + 1, r, y, rc);
  djs.undo();
  for (int i = l; i <= m; ++i) cnt[qr[i].first]++;
}
```

9.4 Manhattan Distance MST

```
void solve(int n) {
  init();
  vector<int> v(n), ds;
  for (int i = 0; i < n; ++i) {
    v[i] = i;
    ds.push_back(x[i] - y[i]);
  }
  sort(ds.begin(), ds.end());
  ds.resize(unique(ds.begin(), ds.end()) - ds.begin());
  sort(v.begin(), v.end(), [&](int i, int j) { return x[i] == x
      [j] ? y[i] > y[j] : x[i] > x[j]; });
  int j = 0;
  for (int i = 0; i < n; ++i) {
    int p = lower_bound(ds.begin(), ds.end(), x[v[i]] - y[v[i
       ]]) - ds.begin() + 1;
    pair<int, int> q = query(p);
    // query return prefix minimum
    if (~q.second) add_edge(v[i], q.second);
    add(p, make_pair(x[v[i]] + y[v[i]], v[i]));
  }
}

void make_graph() {
  solve(n);
  for (int i = 0; i < n; ++i) swap(x[i], y[i]);
  solve(n);
  for (int i = 0; i < n; ++i) x[i] = -x[i];
  solve(n);
  for (int i = 0; i < n; ++i) swap(x[i], y[i]);
  solve(n);
}
```

9.5 Matroid Intersection

- $x \to y$ if $S - \{x\} \cup \{y\} \in I_1$ with $cost(\{y\})$.
- $source \to y$ if $S \cup \{y\} \in I_1$ with $cost(\{y\})$.
- $y \to x$ if $S - \{x\} \cup \{y\} \in I_2$ with $-cost(\{y\})$.
- $y \to sink$ if $S \cup \{y\} \in I_2$ with $-cost(\{y\})$.

Augmenting path is shortest path from source to sink.

9.6 Divide into O(log) Segments

```
auto get = [&](i64 l, i64 r) {
    vector<pair<i64, i64>> res;
    if (l == 0) {
        i64 high = 1;
        while (i128(high) * 2 <= r) {
            high *= 2;
        }
        res.emplace_back(0, high - 1);
        l = high;
    }
    while (l <= r) {
        i64 nxt = l + lowbit(l) - 1;
        if (nxt > r) {
            for (int b = __builtin_ctzll(l) - 1; b >= 0; --b){
                if (l + (1ll << b) - 1 <= r){
                    res.emplace_back(l, l + (1ll << b) - 1);
                    l += 1ll << b;
                }
            }
            break;
        }
        else {
            res.emplace_back(l, nxt);
            l = nxt + 1;
        }
    }
    return res;
};
vector<pair<i64, i64>> all;
for (auto [l1, r1] : sega) {
    for (auto [l2, r2] : segb) {
        i64 length_1 = __lg(r1 - l1 + 1), length_2 = __lg(r2 -
            l2 + 1);

        i64 length = max(length_1, length_2);

        i64 common_prefix = ((l1 ^ l2) >> length) << length;

        i64 L = common_prefix, R = common_prefix + (1ll <<
            length) - 1;
        all.emplace_back(L, R);
    }
}
```

9.7 unorganized

```
const int N = 1021;
struct CircleCover {
  int C;
  Cir c[N];
  bool g[N][N], overlap[N][N];
  // Area[i] : area covered by at least i circles
  double Area[ N ];
  void init(int _C){ C = _C;}
  struct Teve {
    pdd p; double ang; int add;
    Teve() {}
    Teve(pdd _a, double _b, int _c):p(_a), ang(_b), add(_c){}
    bool operator<(const Teve &a)const
    {return ang < a.ang;}
  }eve[N * 2];
  // strict: x = 0, otherwise x = -1
  bool disjuct(Cir &a, Cir &b, int x)
  {return sign(abs(a.O - b.O) - a.R - b.R) > x;}
  bool contain(Cir &a, Cir &b, int x)
  {return sign(a.R - b.R - abs(a.O - b.O)) > x;}
  bool contain(int i, int j) {
    /* c[j] is non-strictly in c[i]. */
    return (sign(c[i].R - c[j].R) > 0 || (sign(c[i].R - c[j].R)
         == 0 && i < j)) && contain(c[i], c[j], -1);
  }
  void solve(){
    fill_n(Area, C + 2, 0);
    for(int i = 0; i < C; ++i)
      for(int j = 0; j < C; ++j)
        overlap[i][j] = contain(i, j);
    for(int i = 0; i < C; ++i)
      for(int j = 0; j < C; ++j)
        g[i][j] = !(overlap[i][j] || overlap[j][i] ||
            disjuct(c[i], c[j], -1));
    for(int i = 0; i < C; ++i){
      int E = 0, cnt = 1;
      for(int j = 0; j < C; ++j)
        if(j != i && overlap[j][i])
          ++cnt;
      for(int j = 0; j < C; ++j)
        if(i != j && g[i][j]) {
          pdd aa, bb;
          CCinter(c[i], c[j], aa, bb);
          double A = atan2(aa.Y - c[i].O.Y, aa.X - c[i].O.X);
          double B = atan2(bb.Y - c[i].O.Y, bb.X - c[i].O.X);
          eve[E++] = Teve(bb, B, 1), eve[E++] = Teve(aa, A, -1)
              ;
          if(B > A) ++cnt;
        }
      if(E == 0) Area[cnt] += pi * c[i].R * c[i].R;
      else{
        sort(eve, eve + E);
        eve[E] = eve[0];
        for(int j = 0; j < E; ++j){
          cnt += eve[j].add;
          Area[cnt] += cross(eve[j].p, eve[j + 1].p) * .5;
          double theta = eve[j + 1].ang - eve[j].ang;
          if (theta < 0) theta += 2. * pi;
          Area[cnt] += (theta - sin(theta)) * c[i].R * c[i].R *
              .5;
        }
      }
    }
  }
};

double ConvexHullDist(vector<pdd> A, vector<pdd> B) {
    for (auto &p : B) p = {-p.X, -p.Y};
    auto C = Minkowski(A, B); // assert SZ(C) > 0
    if (PointInConvex(C, pdd(0, 0))) return 0;
    double ans = PointSegDist(C.back(), C[0], pdd(0, 0));
    for (int i = 0; i + 1 < SZ(C); ++i) {
        ans = min(ans, PointSegDist(C[i], C[i + 1], pdd(0, 0)))
            ;
    }
    return ans;
}

void rotatingSweepLine(vector<pii> &ps) {
  int n = SZ(ps), m = 0;
  vector<int> id(n), pos(n);
  vector<pii> line(n * (n - 1));
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      if (i != j) line[m++] = pii(i, j);
  sort(ALL(line), [&](pii a, pii b) {
    return cmp(ps[a.Y] - ps[a.X], ps[b.Y] - ps[b.X]);
  }); // cmp(): polar angle compare
  iota(ALL(id), 0);
  sort(ALL(id), [&](int a, int b) {
    if (ps[a].Y != ps[b].Y) return ps[a].Y < ps[b].Y;
    return ps[a] < ps[b];
  }); // initial order, since (1, 0) is the smallest
  for (int i = 0; i < n; ++i) pos[id[i]] = i;
  for (int i = 0; i < m; ++i) {
    auto l = line[i];
    // do something
    tie(pos[l.X], pos[l.Y], id[pos[l.X]], id[pos[l.Y]]) =
        make_tuple(pos[l.Y], pos[l.X], l.Y, l.X);
  }
}

bool PointInConvex(const vector<pll> &C, pll p, bool strict =
     true) {
  int a = 1, b = SZ(C) - 1, r = !strict;
  if (SZ(C) == 0) return false;
  if (SZ(C) < 3) return r && btw(C[0], C.back(), p);
  if (ori(C[0], C[a], C[b]) > 0) swap(a, b);
  if (ori(C[0], C[a], p) >= r || ori(C[0], C[b], p) <= -r)
    return false;
  while (abs(a - b) > 1) {
    int c = (a + b) / 2;
    (ori(C[0], C[c], p) > 0 ? b : a) = c;
  }
  return ori(C[a], C[b], p) < r;
}

llf rat(P a, P b) { return sgn(RE(b)) ? llf(RE(a))/RE(b) : llf(
     IM(a))/IM(b); }
llf polyUnion(vector<vector<P>>& poly) {
  llf ret = 0; // area of poly[i] must be non-negative
  rep(i,0,sz(poly)) rep(v,0,sz(poly[i])) {
    P A = poly[i][v], B = poly[i][(v + 1) % sz(poly[i])];
    vector<pair<llf, int>> segs{{0, 0}, {1, 0}};
    rep(j,0,sz(poly)) if (i != j) {
      rep(u,0,sz(poly[j])) {
```

```
      P C = poly[j][u], D = poly[j][(u + 1) % sz(poly[j])];
      if (int sc = ori(A, B, C), sd = ori(A, B, D); sc != sd)
          {
        llf sa = cross(D-C, A-C), sb = cross(D-C, B-C);
        if (min(sc, sd) < 0)
          segs.emplace_back(sa / (sa - sb), sgn(sc - sd));
      } else if (!sc && !sd && j<i && sgn(dot(B-A,D-C))>0){
        segs.emplace_back(rat(C - A, B - A), 1);
        segs.emplace_back(rat(D - A, B - A), -1);
      }
    }
  }
  sort(segs.begin(), segs.end());
  for (auto &s : segs) s.first = clamp<llf>(s.first, 0, 1);
  llf sum = 0;
  int cnt = segs[0].second;
  rep(j,1,sz(segs)) {
    if (!cnt) sum += segs[j].first - segs[j - 1].first;
    cnt += segs[j].second;
  }
  ret += cross(A,B) * sum;
 }
 return ret / 2;
}

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

template <typename F, typename C> class MCMF {
  static constexpr F INF_F = numeric_limits<F>::max();
  static constexpr C INF_C = numeric_limits<C>::max();
  vector<tuple<int, int, F, C>> es;
  vector<vector<int>> g;
  vector<F> f;
  vector<C> d;
  vector<int> pre, inq;
  void spfa(int s) {
    fill(inq.begin(), inq.end(), 0);
    fill(d.begin(), d.end(), INF_C);
    fill(pre.begin(), pre.end(), -1);

    queue<int> q;
    d[s] = 0;
    q.push(s);
    while (!q.empty()) {
      int u = q.front();
      inq[u] = false;
      q.pop();
      for (int j : g[u]) {
        int to = get<1>(es[j]);
        C w = get<3>(es[j]);
        if (f[j] == 0 || d[to] <= d[u] + w)
          continue;
        d[to] = d[u] + w;
        pre[to] = j;
        if (!inq[to]) {
          inq[to] = true;
          q.push(to);
        }
      }
    }
  }

public:
  MCMF(int n) : g(n), pre(n), inq(n) {}
  void add_edge(int s, int t, F c, C w) {
    g[s].push_back(es.size());
    es.emplace_back(s, t, c, w);
    g[t].push_back(es.size());
    es.emplace_back(t, s, 0, -w);
  }
  pair<F, C> solve(int s, int t, C mx = INF_C / INF_F) {
    add_edge(t, s, INF_F, -mx);
    f.resize(es.size()), d.resize(es.size());
    for (F I = INF_F ^ (INF_F / 2); I; I >>= 1) {
      for (auto &fi : f)
        fi *= 2;
      for (size_t i = 0; i < f.size(); i += 2) {
        auto [u, v, c, w] = es[i];
        if ((c & I) == 0)
          continue;
        if (f[i]) {
          f[i] += 1;
          continue;
        }
        spfa(v);
        if (d[u] == INF_C || d[u] + w >= 0) {
          f[i] += 1;
          continue;
        }
        f[i + 1] += 1;
        while (u != v) {
          int x = pre[u];
          f[x] -= 1;
          f[x ^ 1] += 1;
          u = get<0>(es[x]);
        }
      }
    }

    C w = 0;
    for (size_t i = 1; i + 2 < f.size(); i += 2)
      w -= f[i] * get<3>(es[i]);
    return {f.back(), w};
  }
};

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  int n, m, s, t;
  cin >> n >> m >> s >> t;
  s -= 1, t -= 1;
  MCMF<int64_t, int64_t> mcmf(n);
  for (int i = 0; i < m; ++i) {
    int u, v, f, c;
    cin >> u >> v >> f >> c;
    u -= 1, v -= 1;
    mcmf.add_edge(u, v, f, c);
  }
  auto [f, c] = mcmf.solve(s, t, 1e12);
  cout << f << ' ' << c << '\n';
  return 0;
}

struct WeightGraph {
  static const int inf = INT_MAX;
  static const int maxn = 514;
  struct edge {
    int u, v, w;
    edge(){}
    edge(int u, int v, int w): u(u), v(v), w(w) {}
  };
  int n, n_x;
  edge g[maxn * 2][maxn * 2];
  int lab[maxn * 2];
  int match[maxn * 2], slack[maxn * 2], st[maxn * 2], pa[maxn *
       2];
  int flo_from[maxn * 2][maxn + 1], S[maxn * 2], vis[maxn * 2];
  vector<int> flo[maxn * 2];
  queue<int> q;
  int e_delta(const edge &e) { return lab[e.u] + lab[e.v] - g[e
      .u][e.v].w * 2; }
  void update_slack(int u, int x) { if (!slack[x] || e_delta(g[
      u][x]) < e_delta(g[slack[x]][x])) slack[x] = u; }
  void set_slack(int x) {
    slack[x] = 0;
    for (int u = 1; u <= n; ++u)
      if (g[u][x].w > 0 && st[u] != x && S[st[u]] == 0)
        update_slack(u, x);
  }
  void q_push(int x) {
    if (x <= n) q.push(x);
    else for (size_t i = 0; i < flo[x].size(); i++) q_push(flo[
        x][i]);
  }
  void set_st(int x, int b) {
    st[x] = b;
    if (x > n) for (size_t i = 0; i < flo[x].size(); ++i)
        set_st(flo[x][i], b);
  }
  int get_pr(int b, int xr) {
    int pr = find(flo[b].begin(), flo[b].end(), xr) - flo[b].
        begin();
    if (pr % 2 == 1) {
      reverse(flo[b].begin() + 1, flo[b].end());
      return (int)flo[b].size() - pr;
    }
    return pr;
  }
  void set_match(int u, int v) {
    match[u] = g[u][v].v;
    if (u <= n) return;
    edge e = g[u][v];
    int xr = flo_from[u][e.u], pr = get_pr(u, xr);
```

```
    for (int i = 0; i < pr; ++i) set_match(flo[u][i], flo[u][i
         ^ 1]);
    set_match(xr, v);
    rotate(flo[u].begin(), flo[u].begin() + pr, flo[u].end());
  }
  void augment(int u, int v) {
    for (; ; ) {
      int xnv = st[match[u]];
      set_match(u, v);
      if (!xnv) return;
      set_match(xnv, st[pa[xnv]]);
      u = st[pa[xnv]], v = xnv;
    }
  }
  int get_lca(int u, int v) {
    static int t = 0;
    for (++t; u || v; swap(u, v)) {
      if (u == 0) continue;
      if (vis[u] == t) return u;
      vis[u] = t;
      u = st[match[u]];
      if (u) u = st[pa[u]];
    }
    return 0;
  }
  void add_blossom(int u, int lca, int v) {
    int b = n + 1;
    while (b <= n_x && st[b]) ++b;
    if (b > n_x) ++n_x;
    lab[b] = 0, S[b] = 0;
    match[b] = match[lca];
    flo[b].clear();
    flo[b].push_back(lca);
    for (int x = u, y; x != lca; x = st[pa[y]])
      flo[b].push_back(x), flo[b].push_back(y = st[match[x]]),
           q_push(y);
    reverse(flo[b].begin() + 1, flo[b].end());
    for (int x = v, y; x != lca; x = st[pa[y]])
      flo[b].push_back(x), flo[b].push_back(y = st[match[x]]),
           q_push(y);
    set_st(b, b);
    for (int x = 1; x <= n_x; ++x) g[b][x].w = g[x][b].w = 0;
    for (int x = 1; x <= n; ++x) flo_from[b][x] = 0;
    for (size_t i = 0; i < flo[b].size(); ++i) {
      int xs = flo[b][i];
      for (int x = 1; x <= n_x; ++x)
        if (g[b][x].w == 0 || e_delta(g[xs][x]) < e_delta(g[b][
             x]))
          g[b][x] = g[xs][x], g[x][b] = g[x][xs];
      for (int x = 1; x <= n; ++x)
        if (flo_from[xs][x]) flo_from[b][x] = xs;
    }
    set_slack(b);
  }
  void expand_blossom(int b) {
    for (size_t i = 0; i < flo[b].size(); ++i)
      set_st(flo[b][i], flo[b][i]);
    int xr = flo_from[b][g[b][pa[b]].u], pr = get_pr(b, xr);
    for (int i = 0; i < pr; i += 2) {
      int xs = flo[b][i], xns = flo[b][i + 1];
      pa[xs] = g[xns][xs].u;
      S[xs] = 1, S[xns] = 0;
      slack[xs] = 0, set_slack(xns);
      q_push(xns);
    }
    S[xr] = 1, pa[xr] = pa[b];
    for (size_t i = pr + 1; i < flo[b].size(); ++i) {
      int xs = flo[b][i];
      S[xs] = -1, set_slack(xs);
    }
    st[b] = 0;
  }
  bool on_found_edge(const edge &e) {
    int u = st[e.u], v = st[e.v];
    if (S[v] == -1) {
      pa[v] = e.u, S[v] = 1;
      int nu = st[match[v]];
      slack[v] = slack[nu] = 0;
      S[nu] = 0, q_push(nu);
    } else if (S[v] == 0) {
      int lca = get_lca(u, v);
      if (!lca) return augment(u,v), augment(v,u), true;
      else add_blossom(u, lca, v);
    }
    return false;
  }
  bool matching() {
    memset(S + 1, -1, sizeof(int) * n_x);
    memset(slack + 1, 0, sizeof(int) * n_x);
    q = queue<int>();
    for (int x = 1; x <= n_x; ++x)
      if (st[x] == x && !match[x]) pa[x] = 0, S[x] = 0, q_push(
           x);
    if (q.empty()) return false;
    for (; ; ) {
      while (q.size()) {
        int u = q.front(); q.pop();
        if (S[st[u]] == 1) continue;
        for (int v = 1; v <= n; ++v)
          if (g[u][v].w > 0 && st[u] != st[v]) {
            if (e_delta(g[u][v]) == 0) {
              if (on_found_edge(g[u][v])) return true;
            } else update_slack(u, st[v]);
          }
      }
      int d = inf;
      for (int b = n + 1; b <= n_x; ++b)
        if (st[b] == b && S[b] == 1) d = min(d, lab[b] / 2);
      for (int x = 1; x <= n_x; ++x)
        if (st[x] == x && slack[x]) {
          if (S[x] == -1) d = min(d, e_delta(g[slack[x]][x]));
          else if (S[x] == 0) d = min(d, e_delta(g[slack[x]][x
               ]) / 2);
        }
      for (int u = 1; u <= n; ++u) {
        if (S[st[u]] == 0) {
          if (lab[u] <= d) return 0;
          lab[u] -= d;
        } else if (S[st[u]] == 1) lab[u] += d;
      }
      for (int b = n + 1; b <= n_x; ++b)
        if (st[b] == b) {
          if (S[st[b]] == 0) lab[b] += d * 2;
          else if (S[st[b]] == 1) lab[b] -= d * 2;
        }
      q = queue<int>();
      for (int x = 1; x <= n_x; ++x)
        if (st[x] == x && slack[x] && st[slack[x]] != x &&
             e_delta(g[slack[x]][x]) == 0)
          if (on_found_edge(g[slack[x]][x])) return true;
      for (int b = n + 1; b <= n_x; ++b)
        if (st[b] == b && S[b] == 1 && lab[b] == 0)
             expand_blossom(b);
    }
    return false;
  }
  pair<long long, int> solve() {
    memset(match + 1, 0, sizeof(int) * n);
    n_x = n;
    int n_matches = 0;
    long long tot_weight = 0;
    for (int u = 0; u <= n; ++u) st[u] = u, flo[u].clear();
    int w_max = 0;
    for (int u = 1; u <= n; ++u)
      for (int v = 1; v <= n; ++v) {
        flo_from[u][v] = (u == v ? u : 0);
        w_max = max(w_max, g[u][v].w);
      }
    for (int u = 1; u <= n; ++u) lab[u] = w_max;
    while (matching()) ++n_matches;
    for (int u = 1; u <= n; ++u)
      if (match[u] && match[u] < u)
        tot_weight += g[u][match[u]].w;
    return make_pair(tot_weight, n_matches);
  }
  void add_edge(int ui, int vi, int wi) { g[ui][vi].w = g[vi][
       ui].w = wi; }
  void init(int _n) {
    n = _n;
    for (int u = 1; u <= n; ++u)
      for (int v = 1; v <= n; ++v)
        g[u][v] = edge(u, v, 0);
  }
};
```