河南大學

Henan University

Dissonant Interval

Contestant

曹家宁 罗施达 孙小文

Jianing Cao **Shida Luo** **Xiaowen Sun**

目录

1 图论

1.1 EulerianPath

```cpp
struct Edge {
    bool flag;
    i32 v, pos;
    bool operator < (const Edge &u) const {
        return v < u.v;
    }
};

auto hierholzer(const std::vector<std::vector<i32>> &adj) {
    i32 n = adj.size();
    std::vector<std::vector<Edge>> nadj(n);
    for (i32 u = 0; u < n; ++u) {
        for (const auto &v : adj[u]) {
            nadj[u].push_back({true, v, 0});
        }
        std::sort(nadj[u].begin(), nadj[u].end());
    }
    std::vector<i32> top(n), deg(n);
    for (i32 u = 0; u < n; ++u) {
        for (auto &[flag, v, pos] : nadj[u]) {
            pos = top[v]++;
        }
        deg[u] = adj[u].size();
    }
    std::vector<i32> cnt(n), res;
    auto dfs = [&](auto self, i32 u) -> void {
        i32 cur = cnt[u];
        while (cur < adj[u].size()) {
            auto [flag, v, pos] = nadj[u][cur];
            if (flag) {
                nadj[u][cur].flag = false;
                nadj[v][pos].flag = false;
                cur++;
                self(self, v);
            } else {
                cur++;
            }
        }
        res.push_back(u);
    };
    i32 st = 0;
    for (i32 i = 0; i < n; ++i) {
        if (!deg[st] && deg[i]) {
            st = i;
        } else if (!(deg[st] & 1) && (deg[i] & 1)) {
            st = i;
        }
    }
    dfs(dfs, st);
    std::reverse(res.begin(), res.end());
    return res;
}
```

1.2 Ex-Kruskal

```cpp
struct DSU {
    std::vector<i32> p, siz;
    DSU() {}
    DSU(i32 n) {
        init(n);
    }
    void init(i32 n) {
        p.resize(n);
        siz.assign(n, 1);
        std::iota(p.begin(), p.end(), 0);
    }
    i32 find(i32 x) {
        while (x != p[x]) x = p[x] = p[p[x]];
        return x;
    }
    bool same(i32 x, i32 y) {
        return find(x) == find(y);
    }
    bool merge(i32 x, i32 y) {
        x = find(x);
        y = find(y);
        if (x == y) {
            return false;
        } else {
            p[y] = x;
            siz[x] += siz[y];
            return true;
        }
    }
    i32 size(i32 x) {
        return siz[find(x)];
    }
};

struct HLD {
    i32 n, cur;
    std::vector<i32> siz, top, dep, parent, in, out, seq;
```

```
    std::vector<std::vector<i32>> adj;
    HLD() {}
    HLD(i32 n) {
        init(n);
    }
    void init(i32 n) {
        this->n = n;
        in.resize(n);
        out.resize(n);
        siz.resize(n);
        top.resize(n);
        dep.resize(n);
        seq.resize(n);
        parent.resize(n);
        adj.assign(n, {});
        cur = 0;
    }
    void addEdge(i32 u, i32 v) {
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    void work(i32 root = 0) {
        top[root] = root;
        dep[root] = 0;
        parent[root] = -1;
        dfs1(root);
        dfs2(root);
    }
    void dfs1(i32 u) {
        if (parent[u] != -1) {
            adj[u].erase(std::find(adj[u].begin(), adj[u].end(), parent[u]));
        }
        siz[u] = 1;
        for (auto &v : adj[u]) {
            parent[v] = u;
            dep[v] = dep[u] + 1;
            dfs1(v);
            siz[u] += siz[v];
            if (siz[v] > siz[adj[u][0]]) {
                std::swap(v, adj[u][0]);
            }
        }
    }
    void dfs2(i32 u) {
        in[u] = cur++;
        seq[in[u]] = u;
        for (auto v : adj[u]) {
            top[v] = v == adj[u][0] ? top[u] : v;
            dfs2(v);
        }
        out[u] = cur;
    }
    i32 lca(i32 u, i32 v) {
        while (top[u] != top[v]) {
            if (dep[top[u]] > dep[top[v]]) {
                u = parent[top[u]];
            } else {
                v = parent[top[v]];
            }
        }
        return dep[u] < dep[v] ? u : v;
    }
    i32 rootedLca(i32 a, i32 b, i32 c) {
        return lca(a, b) ^ lca(b, c) ^ lca(c, a);
    }
};

struct Node {
    i32 u, v;
    i64 w;
    bool operator < (const Node &u) const {
        return w < u.w;
    }
    bool operator > (const Node &u) const {
        return w > u.w;
    }
};

template <typename Func = std::function<bool(const Node&, const Node&)>>
struct Kruskal {
    Func cmp;
    i32 n;
    HLD tree;
    DSU copo;
    std::vector<i64> val;
    Kruskal(i32 n, std::vector<Node> &edg, Func cmp) : n(n), cmp(cmp) {
        val.resize(2 * n - 1);
        tree.init(2 * n - 1);
        copo.init(2 * n - 1);
        build(edg);
    }
    void build(std::vector<Node> &edg) {
        std::sort(edg.begin(), edg.end(), cmp);
        i32 m = edg.size();
        i32 cur = n;
        for (i32 i = 0; i < m; ++i) {
```

```
            auto [u, v, w] = edg[i];
            u = copo.find(u);
            v = copo.find(v);
            if (u != v) {
                copo.merge(cur, u);
                copo.merge(cur, v);
                tree.addEdge(cur, u);
                tree.addEdge(cur, v);
                val[cur] = w;
                cur++;
                if (cur == 2 * n - 1) {
                    break;
                }
            }
        }
        std::vector<bool> vis(2 * n - 1);
        for (i32 i = 0; i < n; ++i) {
            i32 root = copo.find(i);
            if (!vis[root]) {
                tree.work(root);
                vis[root] = true;
            }
        }
    }
    bool same(i32 u, i32 v) {
        return copo.same(u, v);
    }
    i32 lca(i32 u, i32 v) {
        if (copo.same(u, v)) {
            return tree.lca(u, v);
        } else {
            return -1;
        }
    }
    i64 bnPath(i32 u, i32 v) {
        if (copo.same(u, v)) {
            return val[tree.lca(u, v)];
        } else {
            return -1;
        }
    }
};

using greater = std::greater<Node>;
using less = std::less<Node>;
```

1.3 MinCircuit

```
const i64 INF = 1E18;

struct Edge {
    i32 u, v;
    i64 w;
};

std::vector<std::vector<i64>> dist;
std::vector<i32> path;

i64 floyd(i32 n, const std::vector<Edge> &edg) {
    std::vector val(n, std::vector<i64>(n, INF));
    dist = std::vector(n, std::vector<i64>(n, INF));
    for (const auto &[u, v, w] : edg) {
        dist[u][v] = std::min(dist[u][v], w);
        dist[v][u] = std::min(dist[v][u], w);
        val[u][v] = val[v][u] = w;
    }
    for (i32 i = 0; i < n; ++i) {
        dist[i][i] = 0;
    }
    std::vector pos(n, std::vector<i64>(n, -1));
    auto dfs = [&](auto self, i32 u, i32 v) -> void {
        if (pos[u][v] == -1) return;
        i32 k = pos[u][v];
        self(self, u, k);
        path.push_back(k);
        self(self, k, v);
    };
    i64 res = INF;
    for (i32 k = 0; k < n; ++k) {
        for (i32 u = 0; u < k; ++u) {
            for (i32 v = 0; v < u; ++v) {
                i64 cur = dist[u][v] + val[u][k] + val[k][v];
                if (res > cur) {
                    res = cur;
                    path.clear();
                    path.push_back(u);
                    path.push_back(k);
                    path.push_back(v);
                    dfs(dfs, v, u);
                }
            }
        }
        for (i32 u = 0; u < n; ++u) {
            for (i32 v = 0; v < n; ++v) {
                i64 cur = dist[u][k] + dist[k][v];
```

```
                if (dist[u][v] > cur) {
                    dist[u][v] = cur;
                    pos[u][v] = k;
                }
            }
        }
    }
    return res >= INF ? -1 : res;
}
```

1.4 TopoSort

```
auto topo(std::vector<std::vector<i32>> &adj) {
    i32 n = adj.size();
    std::vector<i32> in(n);
    for (i32 i = 0; i < n; ++i) {
        for (const auto &u : adj[i]) {
            in[u]++;
        }
    }
    std::vector<i32> res;
    std::queue<i32> q;
    for (i32 i = 0; i < n; ++i) {
        if (in[i] == 0) {
            q.push(i);
        }
    }
    while (!q.empty()) {
        auto u = q.front();
        q.pop();
        res.push_back(u);
        for (const auto &v : adj[u]) {
            in[v]--;
            if (in[v] == 0) {
                q.push(v);
            }
        }
    }
    return res;
}
```

1.5 TreeCentroid

```
std::vector<i32> siz, wei;

auto getCentroid(const std::vector<std::vector<i32>> &adj) {
    i32 n = adj.size();
    siz.assign(n, 1);
    wei.assign(n, 0);
    std::array<i32, 2> cen{-1, -1};
    auto dfs = [&](auto self, i32 u, i32 p) -> void {
        for (const auto &v : adj[u]) {
            if (v == p) continue;
            self(self, v, u);
            wei[u] = std::max(wei[u], wei[v]);
            siz[u] += siz[v];
        }
        wei[u] = std::max(wei[u], n - siz[u]);
        if (wei[u] <= n / 2) {
            cen[cen[0] != -1] = u;
        }
    };
    return cen;
}
```

1.6 TreeDiameter

```
i32 getDiameter(const std::vector<std::vector<i32>> &adj) {
    i32 d = 0;
    i32 n = adj.size();
    std::vector<i32> dp(n);
    auto dfs = [&](auto self, i32 u, i32 p) -> void {
        for (const auto &v : adj[u]) {
            if (v == p) continue;
            self(self, v, u);
            d = std::max(d, dp[u] + dp[v] + 1);
            dp[u] = std::max(dp[u], dp[v] + 1);
        }
    };
    dfs(dfs, 0, -1);
    return d;
}
```

1.7 全源最短路

1.7.1 Floyd

```
const i64 INF = 1E18;

struct Edge {
    i32 u, v;
    i64 w;
};

```

```cpp
std::vector<std::vector<i64>> dist;

void floyd(i32 n, const std::vector<Edge> &edg) {
    dist = std::vector(n, std::vector<i64>(n, INF));
    for (const auto &[u, v, w] : edg) {
        dist[u][v] = std::min(dist[u][v], w);
        dist[v][u] = std::min(dist[v][u], w);
    }
    for (i32 i = 0; i < n; ++i) {
        dist[i][i] = 0;
    }
    for (i32 k = 0; k < n; ++k) {
        for (i32 u = 0; u < n; ++u) {
            for (i32 v = 0; v < n; ++v) {
                dist[u][v] = std::min(dist[u][v], dist[u][k] + dist[k][v]);
            }
        }
    }
}
```

1.7.2 Johnson

```cpp
const i64 INF = 1E18;

struct Node {
    i32 v;
    i64 w;
    bool operator < (const Node &u) const {
        return w > u.w;
    }
};

std::vector<i64> pot;
std::vector<std::vector<i64>> dist;

bool spfa(i32 s, std::vector<std::vector<Node>> &adj) {
    i32 n = adj.size();
    std::vector<bool> vis(n);
    std::vector<i32> cnt(n);
    pot.assign(n, INF);
    vis[s] = true;
    pot[s] = 0;
    std::queue<i32> q;
    q.push(s);
    while (!q.empty()) {
        i32 u = q.front();
        q.pop();
        vis[u] = false;
        for (const auto &[v, w] : adj[u]) {
            if (pot[v] > pot[u] + w) {
                pot[v] = pot[u] + w;
                cnt[v] = cnt[u] + 1;
                if (cnt[v] > n) {
                    return false;
                }
                if (!vis[v]) {
                    vis[v] = true;
                    q.push(v);
                }
            }
        }
    }
    return true;
}

void dijkstra(i32 s, std::vector<std::vector<Node>> &adj) {
    i32 n = adj.size();
    dist[s].assign(n, INF);
    std::vector<bool> vis(n);
    std::priority_queue<Node> pq;
    pq.push({s, 0});
    dist[s][s] = 0;
    while (!pq.empty()) {
        auto [u, cur] = pq.top();
        pq.pop();
        if (vis[u]) continue;
        vis[u] = true;
        for (const auto &[v, w] : adj[u]) {
            i64 nxt = cur + w;
            if (dist[s][v] <= nxt) continue;
            pq.push({v, nxt});
            dist[s][v] = nxt;
        }
    }
}

bool johnson(std::vector<std::vector<Node>> &adj) {
    i32 n = adj.size();
    adj.push_back({});
    for (i32 i = 0; i < n; ++i) {
        adj[n].push_back({i, 0});
    }
    bool flag = spfa(n, adj);
    adj.pop_back();
    if (!flag) {
        return false;
```

```
    }
    for (i32 u = 0; u < n; ++u) {
        for (auto &[v, w] : adj[u]) {
            w = w + pot[u] - pot[v];
        }
    }
    dist.assign(n, std::vector<i64>(n));
    for (i32 i = 0; i < n; ++i) {
        dijkstra(i, adj);
        for (i32 j = 0; j < n; ++j) {
            if (dist[i][j] == INF) continue;
            dist[i][j] += pot[j] - pot[i];
        }
    }
    return true;
}
```

1.8 单源最短路

1.8.1 Dijkstra

```
const i64 INF = 1E18;

struct Node {
    i32 v;
    i64 w;
    bool operator < (const Node &u) const {
        return w > u.w;
    }
};

std::vector<i64> dist;

void dijkstra(i32 s, std::vector<std::vector<Node>> &adj) {
    i32 n = adj.size();
    std::vector<bool> vis(n);
    dist.assign(n, INF);
    std::priority_queue<Node> pq;
    pq.push({s, 0});
    dist[s] = 0;
    while (!pq.empty()) {
        auto [u, cur] = pq.top();
        pq.pop();
        if (vis[u]) continue;
        vis[u] = true;
        for (const auto &[v, w] : adj[u]) {
            i64 nxt = cur + w;
            if (dist[v] <= nxt) continue;
            pq.push({v, nxt});
            dist[v] = nxt;
        }
    }
}
```

1.8.2 SPFA

```
const i64 INF = 1E18;

struct Node {
    i32 v, w;
};

std::vector<bool> vis;
std::vector<i64> dist;
std::vector<i32> cnt;

bool spfa(i32 s, std::vector<std::vector<Node>> &adj) {
    i32 n = adj.size();
    vis.assign(n, false);
    cnt.assign(n, 0);
    dist.assign(n, INF);
    vis[s] = true;
    dist[s] = 0;
    std::queue<i32> q;
    q.push(s);
    while (!q.empty()) {
        i32 u = q.front();
        q.pop();
        vis[u] = false;
        for (const auto &[v, w] : adj[u]) {
            if (dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                cnt[v] = cnt[u] + 1;
                if (cnt[v] > n) {
                    return false;
                }
                if (!vis[v]) {
                    vis[v] = true;
                    q.push(v);
                }
            }
        }
    }
    return true;
}
```

1.9 图的匹配

1.9.1 AugmentingPath

```cpp
const i64 INF = 1E18;

struct Edge {
    i32 u, v;
    i64 cap, flow;
};

struct Dinic {
    i32 n, s, t;
    std::vector<Edge> edg;
    std::vector<i32> dep, cur;
    std::vector<std::vector<i32>> pos;
    Dinic(i32 n) : n(n) {
        dep.resize(n);
        cur.resize(n);
        pos.resize(n);
    }
    void addEdge(i32 u, i32 v, i64 w) {
        edg.push_back({u, v, w, 0});
        edg.push_back({v, u, 0, 0});
        i32 m = edg.size();
        pos[u].push_back(m - 2);
        pos[v].push_back(m - 1);
    }
    i64 bfs() {
        std::vector<bool> vis(n);
        std::queue<i32> q;
        q.push(s);
        dep[s] = 0;
        vis[s] = true;
        while (!q.empty()) {
            i32 now = q.front();
            q.pop();
            for (int i = 0; i < pos[now].size(); i++) {
                auto &[u, v, cap, flow] = edg[pos[now][i]];
                if (!vis[v] && cap > flow) {
                    dep[v] = dep[u] + 1;
                    vis[v] = true;
                    q.push(v);
                }
            }
        }
        return vis[t];
    }
    i64 dfs(i32 now, i64 ctn) {
        if (now == t || ctn == 0) {
            return ctn;
        }
        i64 res = 0;
        for (i32 i = cur[now]; i < pos[now].size(); ++i) {
            auto &[u, v, cap, flow] = edg[pos[now][i]];
            auto &[ru, rv, rcap, rflow] = edg[pos[now][i] ^ 1];
            cur[now] = i;
            if (dep[v] == dep[u] + 1 && cap > flow) {
                i64 aug = dfs(v, std::min(ctn - res, cap - flow));
                if (aug > 0) {
                    res += aug;
                    flow += aug;
                    rflow -= aug;
                    if (res == ctn) {
                        return res;
                    }
                }
            }
        }
        return res;
    }
    i64 maxFlow(i32 s, i32 t) {
        this->s = s;
        this->t = t;
        i64 res = 0;
        while (bfs()) {
            cur.assign(n, 0);
            res += dfs(s, INF);
        }
        return res;
    }
};

struct AugmentingPath : Dinic {
    std::unordered_set<i32> l, r;
    AugmentingPath(i32 n) : Dinic(n + 2) {}
    void addEdge(i32 u, i32 v) {
        Dinic::addEdge(u, v, 1);
        l.insert(u);
        r.insert(v);
    }
    i64 match() {
        i32 s = n - 1;
        i32 t = n - 2;
        for (const auto &u : l) {
            Dinic::addEdge(s, u, 1);
        }
```

```
        for (const auto &u : r) {
            Dinic::addEdge(u, t, 1);
        }
        return Dinic::maxFlow(s, t);
    }
};
```

1.9.2 Kuhn-Munkres

```
const i64 INF = 1E18;

struct KuhnMunkres {
    i32 n, l, r;
    std::queue<i32> q;
    std::vector<i32> mchl, mchr, pre;
    std::vector<i64> labl, labr, slk;
    std::vector<bool> visl, visr;
    std::vector<std::vector<i64>> wei;
    KuhnMunkres(i32 l, i32 r) : l(l), r(r), n(std::max(l, r)) {
        wei.assign(n, std::vector<i64>(n, 0LL));
        mchl.assign(n, -1);
        mchr.assign(n, -1);
        pre.assign(n, -1);
        labl.assign(n, 0);
        labr.assign(n, 0);
        slk.resize(n);
        visl.resize(n);
        visr.resize(n);
    }
    void addEdge(i32 u, i32 v, i64 w) {
        wei[u][v] = std::max<i64>(0LL, w);
    }
    bool check(i32 idx) {
        visl[idx] = true;
        if (mchl[idx] != -1) {
            q.push(mchl[idx]);
            visr[mchl[idx]] = true;
            return false;
        }
        while (idx != -1) {
            mchl[idx] = pre[idx];
            std::swap(idx, mchr[pre[idx]]);
        }
        return true;
    }
    bool bfs(i32 idx) {
        while (!q.empty()) {
            i32 v = q.front();
            q.pop();
            for (i32 u = 0; u < n; ++u) {
                if (visl[u]) continue;
                i64 d = labl[u] + labr[v] - wei[u][v];
                if (slk[u] < d) {
                    continue;
                }
                pre[u] = v;
                if (d > 0) {
                    slk[u] = d;
                } else if (check(u)) {
                    return true;
                }
            }
        }
        return false;
    }
    i64 match() {
        for (i32 i = 0; i < n; ++i) {
            labl[i] = *std::max_element(wei[i].begin(), wei[i].end());
        }
        for (i32 i = 0; i < n; ++i) {
            visl.assign(n, false);
            visr.assign(n, false);
            slk.assign(n, INF);
            while (!q.empty()) {
                q.pop();
            }
            q.push(i);
            visr[i] = true;
            while (true) {
                if (bfs(i)) {
                    break;
                }
                i64 d = INF;
                for (i32 j = 0; j < n; ++j) {
                    if (!visl[j]) {
                        d = std::min(d, slk[j]);
                    }
                }
                for (i32 j = 0; j < n; ++j) {
                    if (visr[j]) {
                        labr[j] -= d;
                    }
                    if (visl[j]) {
                        labl[j] += d;
                    } else {
                        slk[j] -= d;
```

```
                    }
                }
                while (!q.empty()) {
                    q.pop();
                }
                bool flag = false;
                for (i32 j = 0; j < n; ++j) {
                    if (!visl[j] && !slk[j] && check(j)) {
                        flag = true;
                        break;
                    }
                }
                if (flag) {
                    break;
                }
            }
        }
        i64 res = 0;
        for (i32 i = 0; i < n; ++i) {
            if (mchl[i] == -1) {
                continue;
            } else if (wei[i][mchl[i]] > 0) {
                res += wei[i][mchl[i]];
            } else {
                mchl[i] = -1;
            }
        }
        return res;
    }
};
```

1.10 树哈希

1.10.1 TreeHashing-Rooted

```
std::mt19937 rng(std::chrono::steady_clock::now().time_since_epoch().count());
const u64 MASK = rng();

struct RTHashing {
    std::set<u64> trees;
    std::vector<u64> hash;
    RTHashing(std::vector<std::vector<i32>> &adj) {
        hash.assign(adj.size(), 0);
        std::function<void(i32, i32)> work = [&](i32 u, i32 p) {
            for (const auto &v : adj[u]) {
                if (v == p) continue;
                work(v, u);
                hash[u] += shift(hash[v]);
            }
            trees.insert(hash[u]);
        };
        work(0, -1);
    }
    u64 shift(u64 x) {
        x ^= MASK;
        x ^= x << 13;
        x ^= x >> 7;
        x ^= x << 17;
        x ^= MASK;
        return x;
    }
    u64 getHash(i32 idx) {
        return hash[idx];
    }
    i32 count() {
        return trees.size();
    }
};
```

1.10.2 TreeHashing-Unrooted

```
std::mt19937 rng(std::chrono::steady_clock::now().time_since_epoch().count());
const u64 MASK = rng();

struct UTHashing {
    i32 n;
    std::vector<u64> sub, rt;
    std::map<u64, i64> trees;
    std::vector<std::vector<i32>> *adj;
    UTHashing(std::vector<std::vector<i32>> &adj) {
        n = adj.size();
        this->adj = &adj;
        sub.assign(n, 0);
        rt.assign(n, 0);
        std::function<void(i32, i32)> getSub = [&](i32 u, i32 p) {
            for (const auto &v : adj[u]) {
                if (v == p) continue;
                getSub(v, u);
                sub[u] += shift(sub[v]);
            }
        };
        std::function<void(i32, i32)> getRoot = [&](i32 u, i32 p) {
            for (const auto &v : adj[u]) {
                if (v == p) continue;
                rt[v] = sub[v] + shift(rt[u] - shift(sub[v]));
                getRoot(v, u);
```

```
            }
            trees[hash[u]] = rt[u];
        };
        getSub(0, -1);
        rt[0] = sub[0];
        getRoot(0, -1);
    }
    u64 shift(u64 x) {
        x ^= MASK;
        x ^= x << 13;
        x ^= x >> 7;
        x ^= x << 17;
        x ^= MASK;
        return x;
    }
    i64 count(i64 idx) {
        return rt[idx];
    }
    i64 total() {
        i64 res = 0;
        for (const auto &[x, y] : trees) {
            res += y;
        }
        return res;
    }
};
```

1.11 树链剖分

1.11.1 HLD-Edge

```
const i64 INF = 1E18;
const i64 MOD = 1E9 + 7;

struct Node {
    i32 u, v;
    i64 w;
    bool operator < (const Node &u) const {
        return w < u.w;
    }
};

template <typename Info, typename Tag>
struct HLD {
    i32 n, cur;
    std::vector<Tag> tag;
    std::vector<Info> info;
    std::vector<i32> w, id, fa, val, siz, dep, top;
    std::vector<std::vector<i32>> adj;
    std::vector<Node> edg;
    HLD(i32 n) : n(n), cur(0) {
        info.resize(4 << std::__lg(n));
        tag.resize(4 << std::__lg(n));
        adj.assign(n, {});
        fa.resize(n, -1);
        val.resize(n);
        siz.resize(n);
        dep.resize(n);
        top.resize(n);
        id.resize(n);
        w.resize(n);
    }
    void add(i32 p, const Tag &k) {
        info[p].apply(k);
        tag[p].apply(k);
    }
    void update(i32 p, const Tag &k) {
        info[p].apply(k);
        tag[p].apply(k);
    }
    void pull(i32 p) {
        info[p] = info[p << 1] + info[p << 1 | 1];
    }
    void push(i32 p) {
        update(p << 1, tag[p]);
        update(p << 1 | 1, tag[p]);
        tag[p] = Tag();
    }
    void addEdge(i32 u, i32 v, i32 w) {
        adj[u].push_back(v);
        adj[v].push_back(u);
        edg.push_back({u, v, w});
    }
    void work(i32 root = 0) {
        top[root] = root;
        dep[root] = 0;
        fa[root] = -1;
        dfs1(root);
        dfs2(root);
        for (auto &[u, v, w] : edg) {
            i64 p = lca(u, v);
            if (v == p) {
                val[id[u]] = w;
            } else {
                val[id[v]] = w;
            }
```

```
        }
        auto build = [&](auto self, i32 p, i32 l, i32 r) {
            if(r == l) {
                info[p].init(val[l]);
                return;
            }
            i32 m = (l + r) / 2;
            self(self, p << 1, l, m);
            self(self, p << 1 | 1, m + 1, r);
            pull(p);
        };
        build(build, 1, 0, n - 1);
    }
    void dfs1(i32 u) {
        if (fa[u] != -1) {
            adj[u].erase(std::find(adj[u].begin(), adj[u].end(), fa[u]));
        }
        siz[u] = 1;
        for (auto &v : adj[u]) {
            dep[v] = dep[u] + 1;
            fa[v] = u;
            dfs1(v);
            siz[u] += siz[v];
            if (siz[adj[u][0]] < siz[v]) {
                std::swap(v, adj[u][0]);
            }
        }
    }
    void dfs2(i32 u) {
        id[u] = cur;
        val[cur++] = w[u];
        for (const auto &v : adj[u]) {
            if (v == fa[u]) continue;
            top[v] = v == adj[u][0] ? top[u] : v;
            dfs2(v);
        }
    }
    i32 lca(i32 u, i32 v) {
        while (top[u] != top[v]) {
            if (dep[top[u]] > dep[top[v]]) {
                u = fa[top[u]];
            } else {
                v = fa[top[v]];
            }
        }
        return dep[u] < dep[v] ? u : v;
    }
    void rangeApply(i32 p, i32 x, i32 y, i32 l, i32 r, const Tag &k) {
        if (x <= l && y >= r) {
            add(p, k);
            return;
        }
        push(p);
        i32 m = (l + r) / 2;
        if (x <= m) rangeApply(p << 1, x, y, l, m, k);
        if (y > m) rangeApply(p << 1 | 1, x, y, m + 1, r, k);
        pull(p);
    }
    void pathApply(i32 u, i32 v, const Tag &k) {
        while (top[u] != top[v]) {
            if (dep[top[u]] < dep[top[v]]) {
                std::swap(u, v);
            }
            rangeApply(1, id[top[u]], id[u], 0, n - 1, k);
            u = fa[top[u]];
        }
        if (dep[u] < dep[v]) {
            std::swap(u, v);
        }
        if (u != v) {
            rangeApply(1, id[v] + 1, id[u], 0, n - 1, k);
        }
    }
    void treeApply(i32 u, const Tag &k) {
        if (siz[u] > 1) {
            rangeApply(1, id[u] + 1, id[u] + siz[u] - 1, 0, n - 1, k);
        }
    }
    Info rangeQuery(i32 p, i32 x, i32 y, i32 l, i32 r) {
        if (x <= l && y >= r) return info[p];
        Info res{};
        push(p);
        i32 m = (l + r) / 2;
        if (x <= m) res = (res + rangeQuery(p << 1, x, y, l, m));
        if (y > m) res = (res + rangeQuery(p << 1 | 1, x, y, m + 1, r));
        pull(p);
        return res;
    }
    Info pathQuery(i32 u, i32 v) {
        Info res{};
        while (top[u] != top[v]) {
            if (dep[top[u]] < dep[top[v]]) {
                std::swap(u, v);
            }
            res = res + rangeQuery(1, id[top[u]], id[u], 0, n - 1);
            u = fa[top[u]];
```

```
        }
        if (dep[u] < dep[v]) {
            std::swap(u, v);
        }
        if (u != v) {
            res = res + rangeQuery(1, id[v] + 1, id[u], 0, n - 1);
        }
        return res;
    }
    Info treeQuery(i32 u) {
        if (siz[u] > 1) {
            return rangeQuery(1, id[u] + 1, id[u] + siz[u] - 1, 0, n - 1);
        } else {
            return Info{};
        }
    }
};

struct Tag {
    i64 add = 0;
    void apply(const Tag &v) {
        add = add + v.add;
    }
};

struct Info {
    i64 sum = 0, len = 1;
    std::array<i64, 2> mn{INF, INF};
    std::array<i64, 2> mx{-INF, -INF};
    void init(const i64 &x) {
        mn[0] = mx[0] = x;
        sum = x;
    }
    void apply(const Tag &v) {
        sum += len * v.add;
        mn[0] += v.add;
        mx[0] += v.add;
        mn[1] += v.add;
        mx[1] += v.add;
    }
    Info operator + (const Info &a) {
        Info res;
        if (mn[0] < a.mn[0]) {
            res.mn[0] = mn[0];
            res.mn[1] = std::min({mn[1], a.mn[0], a.mn[1]});
        } else if (mn[0] > a.mn[0]) {
            res.mn[0] = a.mn[0];
            res.mn[1] = std::min({mn[0], mn[1], a.mn[1]});
        } else {
            res.mn[0] = mn[0];
            res.mn[1] = std::min(mn[1], a.mn[1]);
        }
        if (mx[0] > a.mx[0]) {
            res.mx[0] = mx[0];
            res.mx[1] = std::max({mx[1], a.mx[0], a.mx[1]});
        } else if (mx[0] < a.mx[0]) {
            res.mx[0] = a.mx[0];
            res.mx[1] = std::max({mx[0], mx[1], a.mx[1]});
        } else {
            res.mx[0] = mx[0];
            res.mx[1] = std::max(mx[1], res.mx[1]);
        }
        res.sum = sum + a.sum;
        res.len = len + a.len;
        return res;
    }
};
```

1.11.2 HLD-Vertex

```
const i64 INF = 1E18;
const i64 MOD = 1E9 + 7;

template <typename Info, typename Tag>
struct HLD {
    i32 n, cur;
    std::vector<Tag> tag;
    std::vector<Info> info;
    std::vector<i32> w, id, fa, val, siz, dep, top;
    std::vector<std::vector<i32>> adj;
    HLD(i32 n) : n(n), cur(0) {
        info.resize(4 << std::__lg(n));
        tag.resize(4 << std::__lg(n));
        adj.assign(n, {});
        fa.resize(n, -1);
        siz.resize(n);
        dep.resize(n);
        top.resize(n);
        val.resize(n);
        id.resize(n);
        w.resize(n);
    }
    void add(i32 p, const Tag &k) {
        info[p].apply(k);
        tag[p].apply(k);
    }
```

```
    void update(i32 p, const Tag &k) {
        info[p].apply(k);
        tag[p].apply(k);
    }
    void pull(i32 p) {
        info[p] = info[p << 1] + info[p << 1 | 1];
    }
    void push(i32 p) {
        update(p << 1, tag[p]);
        update(p << 1 | 1, tag[p]);
        tag[p] = Tag();
    }
    void addEdge(i32 u, i32 v) {
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    void assign(i32 u, i32 w) {
        this->w[u] = w;
    }
    void work(i32 root = 0) {
        top[root] = root;
        dep[root] = 0;
        fa[root] = -1;
        dfs1(root);
        dfs2(root);
        auto build = [&](auto self, i32 p, i32 l, i32 r) {
            if(r == l) {
                info[p].init(val[l]);
                return;
            }
            i32 m = (l + r) / 2;
            self(self, p << 1, l, m);
            self(self, p << 1 | 1, m + 1, r);
            pull(p);
        };
        build(build, 1, 0, n - 1);
    }
    void dfs1(i32 u) {
        if (fa[u] != -1) {
            adj[u].erase(std::find(adj[u].begin(), adj[u].end(), fa[u]));
        }
        siz[u] = 1;
        for (auto &v : adj[u]) {
            dep[v] = dep[u] + 1;
            fa[v] = u;
            dfs1(v);
            siz[u] += siz[v];
            if (siz[adj[u][0]] < siz[v]) {
                std::swap(v, adj[u][0]);
            }
        }
    }
    void dfs2(i32 u) {
        id[u] = cur;
        val[cur++] = w[u];
        for (const auto &v : adj[u]) {
            if (v == fa[u]) continue;
            top[v] = v == adj[u][0] ? top[u] : v;
            dfs2(v);
        }
    }
    i32 lca(i32 u, i32 v) {
        while (top[u] != top[v]) {
            if (dep[top[u]] > dep[top[v]]) {
                u = fa[top[u]];
            } else {
                v = fa[top[v]];
            }
        }
        return dep[u] < dep[v] ? u : v;
    }
    void rangeApply(i32 p, i32 x, i32 y, i32 l, i32 r, const Tag &k) {
        if (x <= l && y >= r) {
            add(p, k);
            return;
        }
        push(p);
        i32 m = (l + r) / 2;
        if (x <= m) rangeApply(p << 1, x, y, l, m, k);
        if (y > m) rangeApply(p << 1 | 1, x, y, m + 1, r, k);
        pull(p);
    }
    void pathApply(i32 u, i32 v, const Tag &k) {
        while (top[u] != top[v]) {
            if (dep[top[u]] < dep[top[v]]) {
                std::swap(u, v);
            }
            rangeApply(1, id[top[u]], id[u], 0, n - 1, k);
            u = fa[top[u]];
        }
        if (dep[u] < dep[v]) {
            std::swap(u, v);
        }
        rangeApply(1, id[v], id[u], 0, n - 1, k);
    }
    void treeApply(i32 u, const Tag &k) {
```

```
        rangeApply(1, id[u], id[u] + siz[u] - 1, 0, n - 1, k);
    }
    Info rangeQuery(i32 p, i32 x, i32 y, i32 l, i32 r) {
        if (x <= l && y >= r) return info[p];
        Info res{};
        push(p);
        i32 m = (l + r) / 2;
        if (x <= m) res = (res + rangeQuery(p << 1, x, y, l, m));
        if (y > m) res = (res + rangeQuery(p << 1 | 1, x, y, m + 1, r));
        pull(p);
        return res;
    }
    Info pathQuery(i32 u, i32 v) {
        Info res{};
        while (top[u] != top[v]) {
            if (dep[top[u]] < dep[top[v]]) {
                std::swap(u, v);
            }
            res = res + rangeQuery(1, id[top[u]], id[u], 0, n - 1);
            u = fa[top[u]];
        }
        if (dep[u] < dep[v]) {
            std::swap(u, v);
        }
        res = res + rangeQuery(1, id[v], id[u], 0, n - 1);
        return res;
    }
    Info treeQuery(i32 u) {
        return rangeQuery(1, id[u], id[u] + siz[u] - 1, 0, n - 1);
    }
};

struct Tag {
    i64 add = 0;
    void apply(const Tag &v) {
        add = add + v.add;
    }
};

struct Info {
    i64 sum = 0, len = 1;
    i64 mn = INF, mx = -INF;
    void init(const i64 &x) {
        mn = mx = x;
        sum = x;
    }
    void apply(const Tag &v) {
        sum += len * v.add;
        mn += v.add;
        mx += v.add;
    }
    Info operator + (const Info &a) {
        Info res;
        res.mn = std::min(mn, a.mn);
        res.mx = std::max(mx, a.mx);
        res.sum = sum + a.sum;
        res.len = len + a.len;
        return res;
    }
};
```

1.11.3 HLD

```
struct HLD {
    i32 n, cur;
    std::vector<i32> siz, top, dep, parent, in, out, seq;
    std::vector<std::vector<i32>> adj;
    HLD() {}
    HLD(i32 n) {
        init(n);
    }
    void init(i32 n) {
        this->n = n;
        in.resize(n);
        out.resize(n);
        siz.resize(n);
        top.resize(n);
        dep.resize(n);
        seq.resize(n);
        parent.resize(n);
        adj.assign(n, {});
        cur = 0;
    }
    void addEdge(i32 u, i32 v) {
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    void work(i32 root = 0) {
        top[root] = root;
        dep[root] = 0;
        parent[root] = -1;
        dfs1(root);
        dfs2(root);
    }
    void dfs1(i32 u) {
        if (parent[u] != -1) {
```

```
            adj[u].erase(std::find(adj[u].begin(), adj[u].end(), parent[u]));
        }
        siz[u] = 1;
        for (auto &v : adj[u]) {
            parent[v] = u;
            dep[v] = dep[u] + 1;
            dfs1(v);
            siz[u] += siz[v];
            if (siz[v] > siz[adj[u][0]]) {
                std::swap(v, adj[u][0]);
            }
        }
    }
    void dfs2(i32 u) {
        in[u] = cur++;
        seq[in[u]] = u;
        for (auto v : adj[u]) {
            top[v] = v == adj[u][0] ? top[u] : v;
            dfs2(v);
        }
        out[u] = cur;
    }
    i32 lca(i32 u, i32 v) {
        while (top[u] != top[v]) {
            if (dep[top[u]] > dep[top[v]]) {
                u = parent[top[u]];
            } else {
                v = parent[top[v]];
            }
        }
        return dep[u] < dep[v] ? u : v;
    }
    i32 dist(i32 u, i32 v) {
        return dep[u] + dep[v] - 2 * dep[lca(u, v)];
    }
    i32 jump(i32 u, i32 k) {
        if (dep[u] < k) {
            return -1;
        }
        i32 d = dep[u] - k;
        while (dep[top[u]] > d) {
            u = parent[top[u]];
        }
        return seq[in[u] - dep[u] + d];
    }
    bool isAncester(i32 u, i32 v) {
        return in[u] <= in[v] && in[v] < out[u];
    }
    i32 rootedParent(i32 u, i32 v) {
        std::swap(u, v);
        if (u == v) {
            return u;
        }
        if (!isAncester(u, v)) {
            return parent[u];
        }
        auto dfnCmp = [&](const i32 &x, const i32 &y) {
            return in[x] < in[y];
        };
        auto it = std::upper_bound(adj[u].begin(), adj[u].end(), v, dfnCmp) -
    1;
        return *it;
    }
    i32 rootedSize(i32 u, i32 v) {
        if (u == v) {
            return n;
        }
        if (!isAncester(v, u)) {
            return siz[v];
        }
        return n - siz[rootedParent(u, v)];
    }
    i32 rootedLca(i32 a, i32 b, i32 c) {
        return lca(a, b) ^ lca(b, c) ^ lca(c, a);
    }
};
```

1.12 环计数

1.12.1 CommonRing

```
i64 countRings(const std::vector<std::vector<i32>> &adj) {
    i32 n = adj.size();
    std::vector dp(1 << n, std::vector<i64>(n));
    for (i32 i = 0; i < n; ++i) {
        dp[1 << i][i] = 1;
    }
    i64 cnt = 0;
    for (i32 i = 0; i < n; ++i) {
        cnt += adj[i].size();
    }
    i64 res = 0;
    for (i32 s = 1; s < (1 << n); ++s) {
        for (i32 u = 0; u < n; ++u) {
            if (!dp[s][u]) continue;
            for (const auto &v : adj[u]) {
```

```
                if ((s & -s) > (1 << v)) {
                    continue;
                }
                if (s & (1 << v)) {
                    if ((s & -s) == (1 << v)) {
                        res += dp[s][u];
                    }
                } else {
                    dp[s | (1 << v)][v] += dp[s][u];
                }
            }
        }
    }
    res = (res - cnt / 2) / 2;
    return res;
}
```

1.12.2 QuarternaryRing

```
i64 countRings(const std::vector<std::vector<i32>> &adj) {
    i32 n = adj.size();
    std::vector<i32> deg(n);
    for (i32 i = 0; i < n; ++i) {
        deg[i] = adj[i].size();
    }
    std::vector<std::vector<i32>> nadj(n);
    for (i32 u = 0; u < n; ++u) {
        for (const auto &v : adj[u]) {
            if (deg[u] == deg[v] && u > v) {
                nadj[u].push_back(v);
            }
            if (deg[u] > deg[v]) {
                nadj[u].push_back(v);
            }
        }
    }
    i64 res = 0;
    std::vector<i32> cnt(n);
    for (i32 u = 0; u < n; ++u) {
        for (const auto &v : nadj[u]) {
            for (const auto &w : adj[v]) {
                if (deg[u] == deg[w] && u <= w) continue;
                if (deg[u] < deg[w]) continue;
                res += cnt[w];
                cnt[w]++;
            }
        }
        for (const auto &v : nadj[u]) {
            for (const auto &w : adj[v]) {
                cnt[w] = 0;
            }
        }
    }
    return res;
}
```

1.12.3 TernaryRing

```
i64 countRings(const std::vector<std::vector<i32>> &adj) {
    i32 n = adj.size();
    std::vector<i32> deg(n);
    for (i32 u = 0; u < n; ++u) {
        for (const auto &v : adj[u]) {
            deg[v]++;
        }
    }
    std::vector<std::vector<i32>> nadj(n);
    for (i32 u = 0; u < n; ++u) {
        for (const auto &v : adj[u]) {
            if (deg[u] == deg[v] && u < v) {
                nadj[u].push_back(v);
            }
            if (deg[u] < deg[v]) {
                nadj[u].push_back(v);
            }
        }
    }
    i64 res = 0;
    std::vector<i32> time(n, -1);
    for (i32 u = 0; u < n; ++u) {
        for (const auto &v : nadj[u]) {
            time[v] = u;
        }
        for (const auto &v : nadj[u]) {
            for (const auto &w : nadj[v]) {
                if (time[w] == u) {
                    res++;
                }
            }
        }
    }
    return res;
}
```

1.13 生成树

1.13.1 DMST

```cpp
const i64 INF = 1E18;

struct DSU {
    std::vector<i32> p, siz;
    DSU() {}
    DSU(i32 n) {
        init(n);
    }
    void init(i32 n) {
        p.resize(n);
        siz.assign(n, 1);
        std::iota(p.begin(), p.end(), 0);
    }
    i32 find(i32 x) {
        while (x != p[x]) x = p[x] = p[p[x]];
        return x;
    }
    bool same(i32 x, i32 y) {
        return find(x) == find(y);
    }
    bool merge(i32 x, i32 y) {
        x = find(x);
        y = find(y);
        if (x == y) {
            return false;
        } else {
            p[y] = x;
            siz[x] += siz[y];
            return true;
        }
    }
    i32 size(i32 x) {
        return siz[find(x)];
    }
};

template <typename Info, typename Tag, typename Func = std::less<Info>>
struct LeftistTree {
    struct Node {
        Info info;
        Tag tag;
        i32 dis = 0;
        Node *lc = nullptr;
        Node *rc = nullptr;
        Node(const Info &x) : info(x) {}
    } *root = nullptr;
    i32 siz = 0;
    Func cmp;
    LeftistTree() = default;
    LeftistTree(Func cmp) : cmp(cmp) {}
    ~LeftistTree() {
        clear();
    }
    void abdicate() {
        root = nullptr;
        siz = 0;
    }
    i32 dist(Node *x) {
        if (x == nullptr) {
            return -1;
        }
        return x->dis;
    }
    void pushdown(Node *x) {
        if (x == nullptr) {
            return;
        }
        x->info.apply(x->tag);
        if (x->lc != nullptr) {
            x->lc->tag.apply(x->tag);
        }
        if (x->rc != nullptr) {
            x->rc->tag.apply(x->tag);
        }
        x->tag = Tag{};
    }
    Node* merge(Node* x, Node* y) {
        if (x == nullptr) return y;
        if (y == nullptr) return x;
        pushdown(x);
        pushdown(y);
        if (cmp(x->info, y->info)) {
            std::swap(x, y);
        }
        x->rc = merge(x->rc, y);
        if (dist(x->lc) < dist(x->rc)) {
            std::swap(x->lc, x->rc);
        }
        x->dis = dist(x->rc) + 1;
        return x;
    }
    void merge(LeftistTree &x) {
        root = merge(root, x.root);
```

```
        siz += x.size();
        x.abdicate();
    }
    void clear() {
        if (root == nullptr) return;
        std::queue<Node*> q;
        q.push(root);
        while (!q.empty()) {
            Node *u = q.front();
            q.pop();
            if (u->lc != nullptr) {
                q.push(u->lc);
            }
            if (u->rc != nullptr) {
                q.push(u->rc);
            }
            delete u;
        }
        abdicate();
    }
    void push(Info x) {
        Node *temp = new Node(x);
        root = merge(root, temp);
        siz++;
    }
    void pop() {
        pushdown(root);
        Node *temp = root;
        root = merge(root->lc, root->rc);
        delete temp;
        siz--;
    }
    void apply(const Tag &v) {
        if (root != nullptr) {
            root->tag.apply(v);
        }
    }
    bool empty() {
        return root == nullptr;
    }
    Info top() {
        pushdown(root);
        return root->info;
    }
    i32 size() {
        return siz;
    }
};
```

```

struct Tag {
    i64 add = 0;
    void apply(const Tag &v) {
        add += v.add;
    }
};

struct Info {
    i32 u, v;
    i64 w;
    void apply(const Tag &v) {
        w += v.add;
    }
    bool operator < (const Info &u) const {
        return w < u.w;
    }
    bool operator > (const Info &u) const {
        return w > u.w;
    }
};

i64 dmst(i32 n, i32 r, const std::vector<Info> &edg) {
    i32 m = edg.size();
    std::vector lt(2 * n, LeftistTree<Info, Tag, std::greater<Info>>(std::
    greater<Info>()));
    for (const auto &e : edg) {
        lt[e.v].push(e);
    }
    for (i32 i = 0; i < n; ++i) {
        i32 u = i;
        i32 v = (i + 1) % n;
        lt[v].push({u, v, INF});
    }
    DSU dsu(2 * n);
    std::vector<i32> stk;
    std::vector<bool> vis(2 * n);
    stk.push_back(r);
    vis[r] = true;
    i64 res = 0;
    i32 cur = n;
    while (!lt[stk.back()].empty()) {
        i32 u = stk.back();
        i32 rt = dsu.find(lt[u].top().u);
        if (rt == u) {
            lt[u].pop();
            continue;
        }
```

```
        if (!vis[rt]) {
            stk.push_back(rt);
            vis[rt] = true;
            continue;
        }
        i32 nu = cur++;
        while (vis[rt]) {
            i32 v = stk.back();
            stk.pop_back();
            vis[v] = false;
            dsu.merge(nu, v);
            auto mn = lt[v].top();
            lt[v].pop();
            lt[v].apply({-mn.w});
            lt[nu].merge(lt[v]);
            if (!dsu.same(mn.v, r)) {
                res += mn.w;
            }
        }
        stk.push_back(nu);
        vis[nu] = true;
    }
    return res >= INF ? -1 : res;
}
```

1.13.2 Kruskal

```
struct DSU {
    std::vector<i32> p, siz;
    DSU() {}
    DSU(i32 n) {
        init(n);
    }
    void init(i32 n) {
        p.resize(n);
        siz.assign(n, 1);
        std::iota(p.begin(), p.end(), 0);
    }
    i32 find(i32 x) {
        while (x != p[x]) x = p[x] = p[p[x]];
        return x;
    }
    bool same(i32 x, i32 y) {
        return find(x) == find(y);
    }
    bool merge(i32 x, i32 y) {
        x = find(x);
        y = find(y);
        if (x == y) {
            return false;
        } else {
            p[y] = x;
            siz[x] += siz[y];
            return true;
        }
    }
    i32 size(i32 x) {
        return siz[find(x)];
    }
};

struct Node {
    i32 u, v;
    i64 w;
    bool operator < (const Node &u) const {
        return w > u.w;
    }
};

i64 kruskal(i32 n, std::vector<Node> &edg) {
    std::priority_queue<Node> pq;
    for (const auto &e : edg) {
        pq.push(e);
    }
    i64 res = 0;
    DSU dsu(n);
    while (!pq.empty()) {
        auto [u, v, w] = pq.top();
        pq.pop();
        if (dsu.merge(u, v)) {
            res += w;
        }
    }
    if (dsu.size(0) != n) {
        return -1;
    }
    return res;
}
```

1.13.3 SSST

```
const i64 INF = 1E18;
const i64 MOD = 1E9 + 7;

struct Node {
    i32 u, v;
```

```
    i64 w;
    bool operator < (const Node &u) const {
        return w < u.w;
    }
};

template <typename Info, typename Tag>
struct HLD {
    i32 n, cur;
    std::vector<Tag> tag;
    std::vector<Info> info;
    std::vector<i32> w, id, fa, val, siz, dep, top;
    std::vector<std::vector<i32>> adj;
    std::vector<Node> edg;
    HLD(i32 n) : n(n), cur(0) {
        info.resize(4 << std::__lg(n));
        tag.resize(4 << std::__lg(n));
        adj.assign(n, {});
        fa.resize(n, -1);
        val.resize(n);
        siz.resize(n);
        dep.resize(n);
        top.resize(n);
        id.resize(n);
        w.resize(n);
    }
    void add(i32 p, const Tag &k) {
        info[p].apply(k);
        tag[p].apply(k);
    }
    void update(i32 p, const Tag &k) {
        info[p].apply(k);
        tag[p].apply(k);
    }
    void pull(i32 p) {
        info[p] = info[p << 1] + info[p << 1 | 1];
    }
    void push(i32 p) {
        update(p << 1, tag[p]);
        update(p << 1 | 1, tag[p]);
        tag[p] = Tag();
    }
    void addEdge(i32 u, i32 v, i32 w) {
        adj[u].push_back(v);
        adj[v].push_back(u);
        edg.push_back({u, v, w});
    }
    void work(i32 root = 0) {
        top[root] = root;
        dep[root] = 0;
        fa[root] = -1;
        dfs1(root);
        dfs2(root);
        for (auto &[u, v, w] : edg) {
            i64 p = lca(u, v);
            if (v == p) {
                val[id[u]] = w;
            } else {
                val[id[v]] = w;
            }
        }
        auto build = [&](auto self, i32 p, i32 l, i32 r) {
            if(r == l) {
                info[p].init(val[l]);
                return;
            }
            i32 m = (l + r) / 2;
            self(self, p << 1, l, m);
            self(self, p << 1 | 1, m + 1, r);
            pull(p);
        };
        build(build, 1, 0, n - 1);
    }
    void dfs1(i32 u) {
        if (fa[u] != -1) {
            adj[u].erase(std::find(adj[u].begin(), adj[u].end(), fa[u]));
        }
        siz[u] = 1;
        for (auto &v : adj[u]) {
            dep[v] = dep[u] + 1;
            fa[v] = u;
            dfs1(v);
            siz[u] += siz[v];
            if (siz[adj[u][0]] < siz[v]) {
                std::swap(v, adj[u][0]);
            }
        }
    }
    void dfs2(i32 u) {
        id[u] = cur;
        val[cur++] = w[u];
        for (const auto &v : adj[u]) {
            if (v == fa[u]) continue;
            top[v] = v == adj[u][0] ? top[u] : v;
            dfs2(v);
        }
```

```
    }
    i32 lca(i32 u, i32 v) {
        while (top[u] != top[v]) {
            if (dep[top[u]] > dep[top[v]]) {
                u = fa[top[u]];
            } else {
                v = fa[top[v]];
            }
        }
        return dep[u] < dep[v] ? u : v;
    }
    void rangeApply(i32 p, i32 x, i32 y, i32 l, i32 r, const Tag &k) {
        if (x <= l && y >= r) {
            add(p, k);
            return;
        }
        push(p);
        i32 m = (l + r) / 2;
        if (x <= m) rangeApply(p << 1, x, y, l, m, k);
        if (y > m) rangeApply(p << 1 | 1, x, y, m + 1, r, k);
        pull(p);
    }
    void pathApply(i32 u, i32 v, const Tag &k) {
        while (top[u] != top[v]) {
            if (dep[top[u]] < dep[top[v]]) {
                std::swap(u, v);
            }
            rangeApply(1, id[top[u]], id[u], 0, n - 1, k);
            u = fa[top[u]];
        }
        if (dep[u] < dep[v]) {
            std::swap(u, v);
        }
        if (u != v) {
            rangeApply(1, id[v] + 1, id[u], 0, n - 1, k);
        }
    }
    void treeApply(i32 u, const Tag &k) {
        rangeApply(1, id[u], id[u] + siz[u] - 1, 0, n - 1, k);
    }
    Info rangeQuery(i32 p, i32 x, i32 y, i32 l, i32 r) {
        if (x <= l && y >= r) return info[p];
        Info res{};
        push(p);
        i32 m = (l + r) / 2;
        if (x <= m) res = (res + rangeQuery(p << 1, x, y, l, m));
        if (y > m) res = (res + rangeQuery(p << 1 | 1, x, y, m + 1, r));
        pull(p);
        return res;
    }
    Info pathQuery(i32 u, i32 v) {
        Info res{};
        while (top[u] != top[v]) {
            if (dep[top[u]] < dep[top[v]]) {
                std::swap(u, v);
            }
            res = res + rangeQuery(1, id[top[u]], id[u], 0, n - 1);
            u = fa[top[u]];
        }
        if (dep[u] < dep[v]) {
            std::swap(u, v);
        }
        if (u != v) {
            res = res + rangeQuery(1, id[v] + 1, id[u], 0, n - 1);
        }
        return res;
    }
    Info treeQuery(i32 u) {
        return rangeQuery(1, id[u], id[u] + siz[u] - 1, 0, n - 1);
    }
};

struct Tag {
    i64 add = 0;
    void apply(const Tag &v) {
        add = add + v.add;
    }
};

struct Info {
    i64 sum = 0, len = 1;
    std::array<i64, 2> mn{INF, INF};
    std::array<i64, 2> mx{-INF, -INF};
    void init(const i64 &x) {
        mn[0] = mx[0] = x;
        sum = x;
    }
    void apply(const Tag &v) {
        sum += len * v.add;
        mn[0] += v.add;
        mx[0] += v.add;
        mn[1] += v.add;
        mx[1] += v.add;
    }
    Info operator + (const Info &a) {
        Info res;
```

```
        if (mn[0] < a.mn[0]) {
            res.mn[0] = mn[0];
            res.mn[1] = std::min({mn[1], a.mn[0], a.mn[1]});
        } else if (mn[0] > a.mn[0]) {
            res.mn[0] = a.mn[0];
            res.mn[1] = std::min({mn[0], mn[1], a.mn[1]});
        } else {
            res.mn[0] = mn[0];
            res.mn[1] = std::min(mn[1], a.mn[1]);
        }
        if (mx[0] > a.mx[0]) {
            res.mx[0] = mx[0];
            res.mx[1] = std::max({mx[1], a.mx[0], a.mx[1]});
        } else if (mx[0] < a.mx[0]) {
            res.mx[0] = a.mx[0];
            res.mx[1] = std::max({mx[0], mx[1], a.mx[1]});
        } else {
            res.mx[0] = mx[0];
            res.mx[1] = std::max(mx[1], res.mx[1]);
        }
        res.sum = sum + a.sum;
        res.len = len + a.len;
        return res;
    }
};

struct DSU {
    std::vector<i32> p, siz;
    DSU() {}
    DSU(i32 n) {
        init(n);
    }
    void init(i32 n) {
        p.resize(n);
        siz.assign(n, 1);
        std::iota(p.begin(), p.end(), 0);
    }
    i32 find(i32 x) {
        while (x != p[x]) x = p[x] = p[p[x]];
        return x;
    }
    bool same(i32 x, i32 y) {
        return find(x) == find(y);
    }
    bool merge(i32 x, i32 y) {
        x = find(x);
        y = find(y);
        if (x == y) {
            return false;
        } else {
            p[y] = x;
            siz[x] += siz[y];
            return true;
        }
    }
    i32 size(i32 x) {
        return siz[find(x)];
    }
};

i64 ssst(i32 n, std::vector<Node> &edg) {
    std::vector<std::vector<i32>> adj(n);
    std::sort(edg.begin(), edg.end());
    i32 m = edg.size();
    i64 res = 0;
    DSU dsu(n);
    HLD<Info, Tag> hld(n);
    std::vector<bool> vis(m);
    for (i32 i = 0; i < m; ++i) {
        auto [u, v, w] = edg[i];
        if (dsu.merge(u, v)) {
            hld.addEdge(u, v, w);
            vis[i] = true;
            res += w;
        }
    }
    hld.work();
    i64 fix = INF;
    for (i32 i = 0; i < m; ++i) {
        if (vis[i]) {
            continue;
        }
        auto [u, v, w] = edg[i];
        if (u == v) {
            continue;
        }
        auto cur = hld.pathQuery(u, v);
        if (w > cur.mx[0]) {
            fix = std::min(fix, w - cur.mx[0]);
        } else if (w > cur.mx[1]) {
            fix = std::min(fix, w - cur.mx[1]);
        }
    }
    res += fix;
    return res;
}
```

1.13.4 SteinerTree

```cpp
const i64 INF = 1E18;

struct Node {
    i32 v, w;
};

std::vector<bool> vis;
std::vector<std::vector<i64>> dp;

void spfa(i32 s, std::vector<std::vector<Node>> &adj) {
    i32 n = adj.size();
    vis.assign(n, false);
    std::queue<i32> q;
    for (i32 i = 0; i < n; ++i) {
        if (dp[s][i] != INF) {
            vis[i] = true;
            q.push(i);
        }
    }
    while (!q.empty()) {
        i32 u = q.front();
        q.pop();
        vis[u] = false;
        for (const auto &[v, w] : adj[u]) {
            if (dp[s][u] + w < dp[s][v]) {
                dp[s][v] = dp[s][u] + w;
                if (!vis[v]) {
                    vis[v] = true;
                    q.push(v);
                }
            }
        }
    }
}

i64 steiner(std::vector<std::vector<Node>> &adj, std::vector<i32> &num) {
    i32 n = adj.size();
    i32 m = num.size();
    dp.assign(1 << m, std::vector<i64>(n, INF));
    for (i32 i = 0; i < m; ++i) {
        dp[1 << i][num[i]] = 0;
    }
    for (i32 s = 1; s < (1 << m); ++s) {
        for (i32 t = (s - 1) & s; t > 0; t = (t - 1) & s) {
            if (t < (s ^ t)) {
                break;
            }
            for (i32 i = 0; i < n; ++i) {
                dp[s][i] = std::min(dp[s][i], dp[t][i] + dp[s ^ t][i]);
            }
        }
        spfa(s, adj);
    }
    return *std::min_element(dp[(1 << m) - 1].begin(), dp[(1 << m) - 1].end());
}
```

1.14 矩阵树定理

1.14.1 MatrixTree-Directed

```cpp
const i64 MOD = 1E9 + 7;

template <typename T>
struct Matrix {
    i32 n, m;
    std::vector<std::vector<T>> v;
    Matrix(i32 n, i32 m) : n(n), m(m) {
        v = std::vector(n, std::vector<T>(m));
    }
    Matrix(const std::vector<std::vector<T>> &v) : Matrix(v.size(), v[0].size()) {
        for (i32 i = 0; i < n; ++i) {
            for (i32 j = 0; j < m; ++j) {
                this->v[i][j] = v[i][j];
            }
        }
    }
    std::vector<T>& operator [] (i32 x) {
        assert(x < n);
        return v[x];
    }
    Matrix<T> operator = (const Matrix<T> &x) {
        n = x.n;
        m = x.m;
        v = x;
        return *this;
    }
    Matrix<T> operator + (const Matrix<T> &x) {
        assert(n == x.n && m == x.m);
        Matrix<T> res(n, m);
        for (i32 i = 0; i < n; ++i) {
            for (i32 j = 0; j < m; ++j) {
                res[i][j] = v[i][j] + v[i][j];
            }
        }
```

```
        return res;
    }
    Matrix<T> operator += (const Matrix<T> &x) {
        return *this = *this + x;
    }
    Matrix<T> operator - (const Matrix<T> &x) {
        assert(n == x.n && m == x.m);
        Matrix<T> res(n, m);
        for (i32 i = 0; i < n; ++i) {
            for (i32 j = 0; j < m; ++j) {
                res[i][j] = v[i][j] - v[i][j];
            }
        }
        return res;
    }
    Matrix<T> operator -= (const Matrix<T> &x) {
        return *this = *this - x;
    }
    Matrix<T> operator * (const Matrix<T> &x) {
        assert(m == x.n);
        Matrix<T> res(n, x.m);
        for (i32 i = 0; i < n; ++i) {
            for (i32 j = 0; j < x.m; ++j) {
                for (i32 k = 0; k < m; ++k) {
                    res[i][j] = res[i][j] + v[i][k] * v[k][j];
                }
            }
        }
        return res;
    }
    Matrix<T> operator *= (const Matrix<T> &x) {
        return *this = *this * x;
    }
    static Matrix<T> power(Matrix<T> a, i64 b) {
        Matrix<T> res(Matrix::eye(a.n));
        while (b > 0) {
            if (b & 1) res = res * a;
            a = a * a;
            b >>= 1;
        }
        return res;
    }
    static Matrix<T> eye(i32 n) {
        Matrix<T> res(n, n);
        for (i32 i = 0; i < n; ++i) {
            res[i][i] = 1;
        }
        return res;
    }
    friend std::ostream& operator << (std::ostream &os, const Matrix<T> &x) {
        for (i32 i = 0; i < x.n; ++i) {
            for (i32 j = 0; j < x.m; ++j) {
                os << x.v[i][j] << " \n"[j + 1 == x.m];
            }
        }
        return os;
    }
};

template <typename T>
i64 det(Matrix<T> x) {
    assert(x.n == x.m);
    i64 res = 1;
    for (i32 i = 0; i < x.n; ++i) {
        i32 pivot = i;
        for (i32 j = i; j < x.n; ++j) {
            if (x[j][i] != 0) {
                pivot = j;
                break;
            }
        }
        if (x[pivot][i] == 0) {
            return 0;
        }
        if (pivot != i) {
            std::swap(x[i], x[pivot]);
            res = (MOD - res) % MOD;
        }
        for (i32 j = i + 1; j < x.n; ++j) {
            if (x[j][i] == 0) continue;
            while (x[i][i] != 0) {
                i64 fix = x[j][i] / x[i][i];
                for (i32 k = i; k < x.n; ++k) {
                    x[j][k] = ((x[j][k] - fix * x[i][k]) % MOD + MOD) % MOD;
                }
                for (i32 k = i; k < x.n; ++k) {
                    std::swap(x[i][k], x[j][k]);
                }
                res = (MOD - res) % MOD;
            }
            for (i32 k = i; k < x.n; ++k) {
                std::swap(x[i][k], x[j][k]);
            }
            res = (MOD - res) % MOD;
        }
        res = res * x[i][i] % MOD;
```

```
    }
    return res;
}

template <typename T>
struct MatrixTree {
    i32 n;
    Matrix<T> root, leaf;
    MatrixTree(const Matrix<T> &adj) : n(adj.n), root(n, n), leaf(n, n) {
        assert(adj.n == adj.m);
        for (i32 i = 0; i < n; ++i) {
            for (i32 j = 0; j < n; ++j) {
                root[i][i] += adj[i][j];
                leaf[i][i] += adj[j][i];
                root[i][j] -= adj[i][j];
                leaf[i][j] -= adj[j][i];
            }
        }
    }
    MatrixTree(const std::vector<std::vector<i32>> &adj) : n(adj.size()), root(n, n), leaf(n, n) {
        for (i32 i = 0; i < n; ++i) {
            root[i][i] = adj.size();
            for (const auto &j : adj[i]) {
                leaf[j][j]++;
                leaf[j][i]--;
                root[i][j]--;
            }
        }
    }
    i64 cntRoot(i32 rt = 0) {
        i32 x = 0, y = 0;
        Matrix<T> sub(n - 1, n - 1);
        for (i32 i = 0; i < n; ++i) {
            if (i == rt) continue;
            for (i32 j = 0; j < n; ++j) {
                if (j == rt) continue;
                sub[x][y] = out[i][j];
                y = (y + 1) % (n - 1);
            }
            x++;
        }
        return det(sub);
    }
    i64 cntLeaf(i32 rt = 0) {
        i32 x = 0, y = 0;
        Matrix<T> sub(n - 1, n - 1);
        for (i32 i = 0; i < n; ++i) {
            if (i == rt) continue;
            for (i32 j = 0; j < n; ++j) {
                if (j == rt) continue;
                sub[x][y] = in[i][j];
                y = (y + 1) % (n - 1);
            }
            x++;
        }
        return det(sub);
    }
};
```

1.14.2 MatrixTree-Undirected

```
const i64 MOD = 1E9 + 7;

i64 power(i64 a, i64 b, i64 p = MOD) {
    i64 res = 1;
    while (b) {
        if (b & 1) res = res * a % p;
        a = a * a % p;
        b >>= 1;
    }
    return res;
}

i64 inv(i64 a, i64 p = MOD) {
    return power(a, p - 2, p);
}

template <typename T>
struct Matrix {
    i32 n, m;
    std::vector<std::vector<T>> v;
    Matrix(i32 n, i32 m) : n(n), m(m) {
        v = std::vector(n, std::vector<T>(m));
    }
    Matrix(const std::vector<std::vector<T>> &v) : Matrix(v.size(), v[0].size()) {
        for (i32 i = 0; i < n; ++i) {
            for (i32 j = 0; j < m; ++j) {
                this->v[i][j] = v[i][j];
            }
        }
    }
    std::vector<T>& operator [] (i32 x) {
        assert(x < n);
        return v[x];
```

```
    }
    Matrix<T> operator = (const Matrix<T> &x) {
        n = x.n;
        m = x.m;
        v = x;
        return *this;
    }
    Matrix<T> operator + (const Matrix<T> &x) {
        assert(n == x.n && m == x.m);
        Matrix<T> res(n, m);
        for (i32 i = 0; i < n; ++i) {
            for (i32 j = 0; j < m; ++j) {
                res[i][j] = v[i][j] + v[i][j];
            }
        }
        return res;
    }
    Matrix<T> operator += (const Matrix<T> &x) {
        return *this = *this + x;
    }
    Matrix<T> operator - (const Matrix<T> &x) {
        assert(n == x.n && m == x.m);
        Matrix<T> res(n, m);
        for (i32 i = 0; i < n; ++i) {
            for (i32 j = 0; j < m; ++j) {
                res[i][j] = v[i][j] - v[i][j];
            }
        }
        return res;
    }
    Matrix<T> operator -= (const Matrix<T> &x) {
        return *this = *this - x;
    }
    Matrix<T> operator * (const Matrix<T> &x) {
        assert(m == x.n);
        Matrix<T> res(n, x.m);
        for (i32 i = 0; i < n; ++i) {
            for (i32 j = 0; j < x.m; ++j) {
                for (i32 k = 0; k < m; ++k) {
                    res[i][j] = res[i][j] + v[i][k] * v[k][j];
                }
            }
        }
        return res;
    }
    Matrix<T> operator *= (const Matrix<T> &x) {
        return *this = *this * x;
    }
    static Matrix<T> power(Matrix<T> a, i64 b) {
        Matrix<T> res(Matrix::eye(a.n));
        while (b > 0) {
            if (b & 1) res = res * a;
            a = a * a;
            b >>= 1;
        }
        return res;
    }
    static Matrix<T> eye(i32 n) {
        Matrix<T> res(n, n);
        for (i32 i = 0; i < n; ++i) {
            res[i][i] = 1;
        }
        return res;
    }
    friend std::ostream& operator << (std::ostream &s, const Matrix<T> &x) {
        for (i32 i = 0; i < x.n; ++i) {
            for (i32 j = 0; j < x.m; ++j) {
                os << x.v[i][j] << " \n"[j + 1 == x.m];
            }
        }
        return os;
    }
};

template <typename T>
i64 det(Matrix<T> x) {
    assert(x.n == x.m);
    i64 res = 1;
    for (i32 i = 0; i < x.n; ++i) {
        i32 pivot = i;
        for (i32 j = i; j < x.n; ++j) {
            if (x[j][i] != 0) {
                pivot = j;
                break;
            }
        }
        if (x[pivot][i] == 0) {
            return 0;
        }
        if (pivot != i) {
            std::swap(x[i], x[pivot]);
            res = (MOD - res) % MOD;
        }
        for (i32 j = i + 1; j < x.n; ++j) {
            if (x[j][i] == 0) continue;
            while (x[i][i] != 0) {
```

```
                i64 fix = x[j][i] / x[i][i];
                for (i32 k = i; k < x.n; ++k) {
                    x[j][k] = ((x[j][k] - fix * x[i][k]) % MOD + MOD) % MOD;
                }
                for (i32 k = i; k < x.n; ++k) {
                    std::swap(x[i][k], x[j][k]);
                }
                res = (MOD - res) % MOD;
            }
            for (i32 k = i; k < x.n; ++k) {
                std::swap(x[i][k], x[j][k]);
            }
            res = (MOD - res) % MOD;
        }
        res = res * x[i][i] % MOD;
    }
    return res;
}

template <typename T>
struct MatrixTree {
    i32 n;
    Matrix<T> laplace;
    MatrixTree(const Matrix<T> &adj) : n(adj.n), laplace(n, n) {
        assert(adj.n == adj.m);
        for (i32 i = 0; i < n; ++i) {
            for (i32 j = 0; j < n; ++j) {
                laplace[i][i] += adj[i][j];
                laplace[i][j] -= adj[i][j];
            }
        }
    }
    MatrixTree(const std::vector<std::vector<T>> &adj) : n(adj.size()), laplace
    (n, n) {
        for (i32 i = 0; i < n; ++i) {
            laplace[i][i] = adj[i].size();
            for (const auto &j : adj[i]) {
                laplace[i][j]--;
            }
        }
    }
    i64 cntTree() {
        Matrix<T> sub(n - 1, n - 1);
        for (i32 i = 0; i < n - 1; ++i) {
            for (i32 j = 0; j < n - 1; ++j) {
                sub[i][j] = laplace[i][j];
            }
        }
        return det(sub);
    }
};
```

1.15 缩点

1.15.1 EBCC

```
std::set<std::pair<i32, i32>> edg;

struct EBCC {
    i32 n;
    std::vector<std::vector<std::pair<i32, i32>>> adj;
    std::vector<i32> dfn, low, bel;
    std::vector<i32> stk;
    i32 cur, cnt, tot;
    EBCC() {}
    EBCC(i32 n) {
        init(n);
    }
    void init(i32 n) {
        this->n = n;
        cur = cnt = tot = 0;
        adj.assign(n, {});
        dfn.assign(n, -1);
        bel.assign(n, -1);
        low.resize(n);
        stk.clear();
    }
    void addEdge(i32 u, i32 v) {
        if (u == v) return;
        adj[u].push_back({v, tot});
        adj[v].push_back({u, tot});
        tot++;
    }
    void dfs(i32 lst, i32 u) {
        dfn[u] = low[u] = cur++;
        stk.push_back(u);
        for (auto &[v, nxt] : adj[u]) {
            if (lst == nxt) continue;
            if (dfn[v] == -1) {
                edg.emplace(u, v);
                dfs(u, v);
                low[u] = std::min(low[u], low[v]);
            } else if (bel[v] == -1 && dfn[v] < dfn[u]) {
                edg.emplace(u, v);
                low[u] = std::min(low[u], dfn[v]);
            }
```

```
        }
        if (dfn[u] == low[u]) {
            i32 v;
            do {
                v = stk.back();
                bel[v] = cnt;
                stk.pop_back();
            } while (v != u);
            cnt++;
        }
    }
    std::vector<i32> work() {
        dfs(-1, 0);
        return bel;
    }
    struct Graph {
        i32 n;
        std::vector<std::pair<i32, i32>> edges;
        std::vector<i32> siz;
        std::vector<i32> cnte;
    };
    Graph compress() {
        Graph g;
        g.n = cnt;
        g.siz.resize(cnt);
        g.cnte.resize(cnt);
        for (i32 i = 0; i < n; ++i) {
            g.siz[bel[i]]++;
            for (auto &[j, idx] : adj[i]) {
                if (bel[i] < bel[j]) {
                    g.edges.emplace_back(bel[i], bel[j]);
                } else if (i < j) {
                    g.cnte[bel[i]]++;
                }
            }
        }
        return g;
    }
};
```

1.15.2 SCC

```
struct SCC {
    i32 n;
    std::vector<std::vector<i32>> adj;
    std::vector<i32> dfn, low, bel;
    std::vector<i32> stk;
    i32 cur, cnt;
    SCC() {}
    SCC(i32 n) {
        init(n);
    }
    void init(i32 n) {
        this->n = n;
        adj.assign(n, {});
        dfn.assign(n, -1);
        bel.assign(n, -1);
        low.resize(n);
        stk.clear();
        cur = cnt = 0;
    }
    void add_edge(i32 u, i32 v) {
        adj[u].push_back(v);
    }
    void dfs(i32 x) {
        dfn[x] = low[x] = cur++;
        stk.push_back(x);
        for (auto y : adj[x]) {
            if (dfn[y] == -1) {
                dfs(y);
                low[x] = std::min(low[x], low[y]);
            } else if (bel[y] == -1) {
                low[x] = std::min(low[x], dfn[y]);
            }
        }
        if (dfn[x] == low[x]) {
            i32 y;
            do {
                y = stk.back();
                bel[y] = cnt;
                stk.pop_back();
            } while (y != x);
            cnt++;
        }
    }
    std::vector<i32> work() {
        for (i32 i = 0; i < n; i++) {
            if (dfn[i] == -1) {
                dfs(i);
            }
        }
        return bel;
    }
};
```

1.16 网络流

1.16.1 Dinic

```cpp
const i64 INF = 1E18;

struct Edge {
    i32 u, v;
    i64 cap, flow;
};

struct Dinic {
    i32 n, s, t;
    std::vector<Edge> edg;
    std::vector<i32> dep, cur;
    std::vector<std::vector<i32>> pos;
    Dinic(i32 n) : n(n) {
        dep.resize(n);
        cur.resize(n);
        pos.resize(n);
    }
    void addEdge(i32 u, i32 v, i64 w) {
        edg.push_back({u, v, w, 0});
        edg.push_back({v, u, 0, 0});
        i32 m = edg.size();
        pos[u].push_back(m - 2);
        pos[v].push_back(m - 1);
    }
    i64 bfs() {
        std::vector<bool> vis(n);
        std::queue<i32> q;
        q.push(s);
        dep[s] = 0;
        vis[s] = true;
        while (!q.empty()) {
            i32 now = q.front();
            q.pop();
            for (int i = 0; i < pos[now].size(); i++) {
                auto &[u, v, cap, flow] = edg[pos[now][i]];
                if (!vis[v] && cap > flow) {
                    dep[v] = dep[u] + 1;
                    vis[v] = true;
                    q.push(v);
                }
            }
        }
        return vis[t];
    }
    i64 dfs(i32 now, i64 ctn) {
        if (now == t || ctn == 0) {
            return ctn;
        }
        i64 res = 0;
        for (i32 i = cur[now]; i < pos[now].size(); ++i) {
            auto &[u, v, cap, flow] = edg[pos[now][i]];
            auto &[ru, rv, rcap, rflow] = edg[pos[now][i] ^ 1];
            cur[now] = i;
            if (dep[v] == dep[u] + 1 && cap > flow) {
                i64 aug = dfs(v, std::min(ctn - res, cap - flow));
                if (aug > 0) {
                    res += aug;
                    flow += aug;
                    rflow -= aug;
                    if (res == ctn) {
                        return res;
                    }
                }
            }
        }
        return res;
    }
    i64 maxFlow(i32 s, i32 t) {
        this->s = s;
        this->t = t;
        i64 res = 0;
        while (bfs()) {
            cur.assign(n, 0);
            res += dfs(s, INF);
        }
        return res;
    }
};
```

1.16.2 ISAP

```cpp
const i64 INF = 1E18;

struct Edge {
    i32 u, v;
    i64 cap, flow;
};

struct ISAP {
    i32 n, s, t;
    std::vector<Edge> edg;
    std::vector<i32> dep, rpos;
    std::vector<std::vector<i32>> pos;
```

```
    ISAP(i32 n) : n(n) {
        rpos.resize(n);
        pos.resize(n);
        dep.resize(n);
    }
    void addEdge(i32 u, i32 v, i64 w) {
        edg.push_back({u, v, w, 0});
        edg.push_back({v, u, 0, 0});
        i32 m = edg.size();
        pos[u].push_back(m - 2);
        pos[v].push_back(m - 1);
    }
    bool bfs() {
        std::vector<bool> vis(n);
        std::queue<i32> q;
        q.push(t);
        dep[t] = 0;
        vis[t] = true;
        while (!q.empty()) {
            i32 now = q.front();
            q.pop();
            for (int i = 0; i < pos[now].size(); i++) {
                auto &[u, v, cap, flow] = edg[pos[now][i] ^ 1];
                if (!vis[u] && cap > flow) {
                    dep[u] = dep[v] + 1;
                    vis[u] = true;
                    q.push(u);
                }
            }
        }
        return vis[s];
    }
    i64 augment() {
        i64 res = INF;
        i32 now = t;
        while (now != s) {
            auto &[u, v, cap, flow] = edg[rpos[now]];
            res = std::min(res, cap - flow);
            now = u;
        }
        now = t;
        while (now != s) {
            edg[rpos[now]].flow += res;
            edg[rpos[now] ^ 1].flow -= res;
            now = edg[rpos[now]].u;
        }
        return res;
    }
    i64 maxFlow(i32 s, i32 t) {
        reset();
        this->s = s;
        this->t = t;
        bfs();
        std::vector<i32> num(n);
        for (i32 i = 0; i < n; ++i) {
            num[dep[i]]++;
        }
        i32 now = s;
        i64 res = 0;
        std::vector<i32> cur(n);
        while (dep[s] < n) {
            if (now == t) {
                res += augment();
                now = s;
            }
            bool flag = false;
            for (i32 i = cur[now]; i < pos[now].size(); ++i) {
                auto &[u, v, cap, flow] = edg[pos[now][i]];
                if (cap > flow && dep[u] == dep[v] + 1) {
                    flag = true;
                    rpos[v] = pos[u][i];
                    cur[u] = i;
                    now = v;
                    break;
                }
            }
            if (!flag) {
                i32 mnd = n - 1;
                for (i32 i = 0; i < pos[now].size(); ++i) {
                    auto &[u, v, cap, flow] = edg[pos[now][i]];
                    if (cap > flow) {
                        mnd = std::min(mnd, dep[v]);
                    }
                }
                if (--num[dep[now]] == 0) {
                    break;
                }
                dep[now] = mnd + 1;
                num[dep[now]]++;
                cur[now] = 0;
                if (now != s) {
                    now = edg[rpos[now]].u;
                }
            }
        }
        return res;
```

```
    }
    void reset() {
        for (auto &e : edg) {
            e.flow = 0;
        }
    }
};
```

1.16.3 Primal-Dual

```
const i64 INF = 1E18;

struct Edge {
    i32 u, v;
    i64 cap, flow, cost;
};

struct PrimalDual {
    i32 n, s, t;
    std::vector<Edge> edg;
    std::vector<i64> pot, dis;
    std::vector<std::pair<i32, i32>> dir;
    std::vector<std::vector<i32>> pos;
    PrimalDual(i32 n) : n(n) {
        dis.resize(n);
        dir.resize(n);
        pot.resize(n);
        pos.resize(n);
    }
    void addEdge(i32 u, i32 v, i64 w, i64 c) {
        edg.push_back({u, v, w, 0, c});
        edg.push_back({v, u, 0, 0, -c});
        i32 m = edg.size();
        pos[u].push_back(m - 2);
        pos[v].push_back(m - 1);
    }
    void spfa() {
        std::vector<bool> vis(n);
        std::queue<i32> q;
        q.push(s);
        pot[s] = 0;
        vis[s] = true;
        while (!q.empty()) {
            i32 now = q.front();
            q.pop();
            vis[now] = false;
            for (i32 i = 0; i < pos[now].size(); ++i) {
                auto &[u, v, cap, flow, cost] = edg[pos[now][i]];
                if (cap - flow && pot[v] > pot[u] + cost) {
                    pot[v] = pot[u] + cost;
                    if (!vis[v]) {
                        vis[v] = true;
                        q.push(v);
                    }
                }
            }
        }
    }
    bool dijkstra() {
        auto cmp = [&](const std::pair<i64, i32> &a,
                       const std::pair<i64, i32> &b) -> bool {
            return a.first > b.first;
        };
        std::priority_queue<std::pair<i64, i32>,
                            std::vector<std::pair<i64, i32>>,
                            decltype(cmp)> pq(cmp);
        std::vector<bool> vis(n);
        dis.assign(n, INF);
        dis[s] = 0;
        pq.push({0, s});
        while (!pq.empty()) {
            i32 now = pq.top().second;
            pq.pop();
            if (vis[now]) {
                continue;
            }
            vis[now] = true;
            for (i32 i = 0; i < pos[now].size(); ++i) {
                auto &[u, v, cap, flow, cost] = edg[pos[now][i]];
                i64 pcost = cost + pot[u] - pot[v];
                if (cap - flow > 0 && dis[v] > dis[u] + pcost) {
                    dir[v] = {u, pos[now][i]};
                    dis[v] = dis[u] + pcost;
                    if (!vis[v]) {
                        pq.push({dis[v], v});
                    }
                }
            }
        }
        return dis[t] < INF;
    }
    std::pair<i64, i64> work(i32 s, i32 t) {
        reset();
        this->s = s;
        this->t = t;
        spfa();
```

```
        i64 mnc = 0;
        i64 mxf = 0;
        while (dijkstra()) {
            i64 aug = INF;
            for (i32 i = 0; i < n; ++i) {
                pot[i] += dis[i];
            }
            for (i32 i = t; i != s; i = dir[i].first) {
                auto &[u, v, cap, flow, cost] = edg[dir[i].second];
                aug = std::min(aug, cap - flow);
            }
            for (i32 i = t; i != s; i = dir[i].first) {
                edg[dir[i].second ^ 1].flow -= aug;
                edg[dir[i].second].flow += aug;
            }
            mnc += aug * pot[t];
            mxf += aug;
        }
        return std::make_pair(mxf, mnc);
    }
    void reset() {
        for (auto &e : edg) {
            e.flow = 0;
        }
        pot.assign(n, INF);
    }
};
```

1.16.4 Stoer-Wagner

```
const i64 INF = 1E18;

struct StoerWagner {
    i32 n;
    std::vector<std::vector<i64>> adj;
    StoerWagner(i32 n) : n(n) {
        adj = std::vector(n, std::vector<i64>(n, 0));
    }
    void addEdge(i32 u, i32 v, i64 w) {
        adj[u][v] += w;
        adj[v][u] += w;
    }
    i64 work() {
        i64 res = INF;
        std::vector<bool> in(n);
        std::vector<i32> bel(n);
        std::iota(bel.begin(), bel.end(), 0);
        for (i32 i = 0; i < n - 1; ++i) {
            std::vector<bool> vis(n);
            std::vector<i64> wei(n);
            i32 lst = -1;
            for (i32 j = 0; j < n - i - 1; ++j) {
                i32 cur = -1;
                for (i32 k = 0; k < n; ++k) {
                    if (!in[k] && !vis[k] && (cur == -1 || wei[k] > wei[cur]))
    {
                        cur = k;
                    }
                }
                vis[cur] = true;
                for (i32 k = 0; k < n; ++k) {
                    if (!in[k] && !vis[k]) {
                        wei[k] += adj[cur][k];
                    }
                }
                lst = cur;
            }
            i32 cur = -1;
            for (i32 k = 0; k < n; ++k) {
                if (!in[k] && !vis[k]) {
                    cur = k;
                    break;
                }
            }
            res = std::min(res, wei[cur]);
            in[cur] = true;
            for (i32 k = 0; k < n; ++k) {
                if (!in[k]) {
                    adj[lst][k] += adj[cur][k];
                    adj[k][lst] += adj[k][cur];
                }
            }
            bel[cur] = lst;
        }
        return res;
    }
};
```

2 字符串

2.1 Duo-Hashing

```
const i64 MOD1 = 212370440130137957;
const i64 MOD2 = 1e9 + 7;
const i64 BASE1 = 127;
```

```
const i64 BASE2 = 131;

i64 power(i64 a, i64 b, i64 p) {
    i64 res = 1;
    while (b) {
        if (b & 1) res = res * a % p;
        a = a * a % p;
        b >>= 1;
    }
    return res;
}

struct Hashing {
    std::vector<i64> h1, h2;
    std::string s;
    Hashing() {}
    Hashing(const std::string &s) : s(s) {
        i32 n = s.length();
        h1.resize(n + 1);
        h2.resize(n + 1);
        for (i32 i = 0; i < n; ++i) {
            h1[i + 1] = (h1[i] * BASE1 + (i64)s[i]) % MOD1;
            h2[i + 1] = (h2[i] * BASE2 + (i64)s[i]) % MOD2;
        }
    }
    auto getHash(i32 l, i32 r) {
        i64 res1 = (h1[r + 1] - h1[l] * power(BASE1, r - l + 1, MOD1) % MOD1 +
    MOD1) % MOD1;
        i64 res2 = (h2[r + 1] - h2[l] * power(BASE2, r - l + 1, MOD2) % MOD2 +
    MOD2) % MOD2;
        return std::make_pair(res1, res2);
    }
    bool hcmp(Hashing &u, i32 l, i32 r) {
        return getHash(l, r) == u.getHash(l, r);
    }
};
```

2.2 KMP

```
std::vector<i32> getPre(std::string s) {
    i32 n = s.length();
    std::vector<i32> res(n);
    for (i32 i = 1; i < n; ++i) {
        i32 j = res[i - 1];
        while (j > 0 && s[i] != s[j]) {
            j = res[j - 1];
        }
        if (s[i] == s[j]) {
            j++;
        }
        res[i] = j;
    }
    return res;
}

std::vector<i32> KMP(std::string p, std::string s) {
    i32 n = p.length();
    i32 m = s.length();
    std::string cur = p + '#' + s;
    std::vector<i32> lps = getPre(cur);
    std::vector<i32> res;
    for (i32 i = n + 1; i <= n + m; ++i) {
        if (lps[i] == n) {
            res.push_back(i - 2 * n);
        }
    }
    return res;
}
```

2.3 Manacher

```
std::vector<i32> manacher(std::string s) {
    std::string t = "#";
    for (auto c : s) {
        t += c;
        t += '#';
    }
    i32 n = t.size();
    std::vector<i32> r(n);
    for (i32 i = 0, j = 0; i < n; ++i) {
        if (2 * j - i >= 0 && j + r[j] > i) {
            r[i] = std::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] += 1;
        }
        if (i + r[i] > j + r[j]) {
            j = i;
        }
    }
    return r;
}
```

2.4 Z-Function

```cpp
std::vector<i32> Z(std::string s) {
    i32 n = s.size();
    std::vector<i32> z(n + 1);
    z[0] = n;
    for (i32 i = 1, j = 1; i < n; i++) {
        z[i] = std::max(0, std::min(j + z[j] - i, z[i - j]));
        while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
            z[i]++;
        }
        if (i + z[i] > j + z[j]) {
            j = i;
        }
    }
    return z;
}
```

3 数学

3.1 CatalanNumbers

```cpp
const i64 MOD = 1E9 + 7;

std::vector<i32> cat;

void catalan(i32 n) {
    cat.resize(n + 1);
    cat[0] = cat[1] = 1;
    for (i32 i = 2; i <= n; ++i) {
        cat[i] = cat[i - 1] * ((4 * i % MOD - 2 + MOD) % MOD) % MOD * inv(i + 1, MOD) % MOD;
    }
}

// catalan-number:
//     H[i] = H[i - 1] * (4n - 2) / (n + 1)
//     H[i] = C(2n, n) / (n + 1)
```

3.2 Combination-Dynamic

```cpp
const i64 MOD = 1E9 + 7;

i64 power(i64 a, i64 b, i64 p = MOD) {
    i64 res = 1;
    while (b) {
        if (b & 1) res = res * a % p;
        a = a * a % p;
        b >>= 1;
    }
    return res;
}

i64 inv(i64 a, i64 p = MOD) {
    return power(a, p - 2, p);
}

template<typename T>
struct Comb {
    i32 n;
    i64 mod = 1E9 + 7;
    std::vector<T> _fac, _invfac, _inv;
    Comb() : n(0) {
        _fac = _invfac = {1};
        _inv = {0};
    }
    Comb(i32 n, i64 mod) : Comb() {
        this->mod = mod;
        init(n);
    }
    void init(i32 m) {
        if (n >= m) return;
        _invfac.resize(m + 1);
        _fac.resize(m + 1);
        _inv.resize(m + 1);
        for (i32 i = n + 1; i <= m; ++i) {
            _fac[i] = _fac[i - 1] * i % mod;
        }
        _invfac[m] = inv(_fac[m], mod);
        for (i32 i = m; i > n; --i) {
            _invfac[i - 1] = _invfac[i] * i % mod;
            _inv[i] = _fac[i - 1] * _invfac[i] % mod;
        }
        n = m;
    }
    T operator () (i32 a,  i32 b) {
        if(a < b || b < 0) return T{};
        else return fac(a) * invfac(a - b) % mod * invfac(b) % mod;
    }
    T invfac(i32 a) {
        if (a > n) init(2 * a);
        return _invfac[a];
    }
    T fac(i32 a) {
        if (a > n) init(2 * a);
        return _fac[a];
```

```
    }
    T civ(i32 a) {
        if (a > n) init(2 * a);
        return _inv[a];
    }
};
```

3.3 Combination

```
const i64 MOD = 1E9 + 7;

template<typename T>
struct Comb {
    T n, mod;
    std::vector<T> fac, inv, invfac;
    Comb(T n = 0, T mod = 1E9 + 7) {
        init(n, mod);
    }
    void init(i32 n, i32 mod) {
        this->n = n;
        this->mod = mod;
        invfac.assign(n, T{});
        inv.assign(n, T{});
        fac.assign(n, T{});
        fac[0] = fac[1] = 1;
        inv[1] = 1;
        invfac[0] = invfac[1] = 1;
        for (i64 i = 2; i < n; ++i) {
            inv[i] = (mod - mod / i) * inv[mod % i] % mod;
            fac[i] = fac[i - 1] * i % mod;
            invfac[i] = invfac[i - 1] * inv[i] % mod;
        }
    }
    T comb(i32 a, i32 b) {
        if(a < b) return T{};
        else return fac[a] * invfac[a - b] % mod * invfac[b] % mod;
    }
};

/*

definition:
    A(n, m) = n! / (n - m)!
    C(n, m) = n! / ((n - m)! * m!)

recursion:
    C(n, m) = C(n, m - 1) * (n - m + 1) / m
    C(n, m) = C(n - 1, m) + C(n - 1, m - 1)

property:
    C(n + m + 1, n) = sigma[i = 0 -> m] { C(n + i, i) }
    C(n, m) * C(m, r) = C(n, r) * C(n - r, m - r)

    sigma[i = 0 -> n] { C(n, i) } = 2 ^ n
    sigma[i = 0 -> n] { C(n, i) * (x ^ i) } = (x + 1) ^ n

    sigma[i = 0 -> n] { (-1) ^ i * C(n, i) } = 0;
    C(n, 0) + C(n, 2) + ... = C(n, 1) + C(n, 3) + ... = 2 ^ (n - 1)

    C(n + m, r) = sigma[i = 0 -> min{n, m, r}] { C(n, i) * C(m, r - i) }

    m * C(n, m) = n * C(n - 1, m - 1)

    sigma[i = 0 -> n] { C(n, i) * i } = n * 2 ^ (n - 1)
    sigma[i = 0 -> n] { C(n, i) * i ^ 2 } = n * (n + 1) * 2 ^ (n - 2)

    sigma[i = 0 -> n] { C(n, i) ^ 2 } = C(2 * n, n)

*/
```

3.4 Matrix

```
const i64 MOD = 998244353;

i64 power(i64 a, i64 b, i64 p = MOD) {
    i64 res = 1;
    while (b) {
        if (b & 1) res = res * a % p;
        a = a * a % p;
        b >>= 1;
    }
    return res;
}

i64 inv(i64 a, i64 p = MOD) {
    return power(a, p - 2, p);
}

template <typename T>
struct Matrix {
    i32 n, m;
    std::vector<std::vector<T>> v;
    Matrix(i32 n, i32 m) : n(n), m(m) {
        v = std::vector(n, std::vector<T>(m));
    }
    Matrix(const std::vector<std::vector<T>> &v) : Matrix(v.size(), v[0].size()
```

```
    ) {
        for (i32 i = 0; i < n; ++i) {
            for (i32 j = 0; j < m; ++j) {
                this->v[i][j] = v[i][j];
            }
        }
    }
    std::vector<T>& operator [] (i32 x) {
        assert(x < n);
        return v[x];
    }
    Matrix<T> operator = (const Matrix<T> &x) {
        n = x.n;
        m = x.m;
        v = x;
        return *this;
    }
    Matrix<T> operator + (const Matrix<T> &x) {
        assert(n == x.n && m == x.m);
        Matrix<T> res(n, m);
        for (i32 i = 0; i < n; ++i) {
            for (i32 j = 0; j < m; ++j) {
                res[i][j] = v[i][j] + v[i][j];
            }
        }
        return res;
    }
    Matrix<T> operator += (const Matrix<T> &x) {
        return *this = *this + x;
    }
    Matrix<T> operator - (const Matrix<T> &x) {
        assert(n == x.n && m == x.m);
        Matrix<T> res(n, m);
        for (i32 i = 0; i < n; ++i) {
            for (i32 j = 0; j < m; ++j) {
                res[i][j] = v[i][j] - v[i][j];
            }
        }
        return res;
    }
    Matrix<T> operator -= (const Matrix<T> &x) {
        return *this = *this - x;
    }
    Matrix<T> operator * (const Matrix<T> &x) {
        assert(m == x.n);
        Matrix<T> res(n, x.m);
        for (i32 i = 0; i < n; ++i) {
            for (i32 j = 0; j < x.m; ++j) {
                for (i32 k = 0; k < m; ++k) {
                    res[i][j] = res[i][j] + v[i][k] * v[k][j];
                }
            }
        }
        return res;
    }
    Matrix<T> operator *= (const Matrix<T> &x) {
        return *this = *this * x;
    }
    static Matrix<T> power(Matrix<T> a, i64 b) {
        Matrix<T> res(Matrix::eye(a.n));
        while (b > 0) {
            if (b & 1) res = res * a;
            a = a * a;
            b >>= 1;
        }
        return res;
    }
    static Matrix<T> eye(i32 n) {
        Matrix<T> res(n, n);
        for (i32 i = 0; i < n; ++i) {
            res[i][i] = 1;
        }
        return res;
    }
    friend std::ostream& operator << (std::ostream &os, const Matrix<T> &x) {
        for (i32 i = 0; i < x.n; ++i) {
            for (i32 j = 0; j < x.m; ++j) {
                os << x.v[i][j] << " \n"[j + 1 == x.m];
            }
        }
        return os;
    }
};

template <typename T>
i64 det(Matrix<T> x) {
    assert(x.n == x.m);
    i64 res = 1;
    for (i32 i = 0; i < x.n; ++i) {
        i32 pivot = i;
        for (i32 j = i; j < x.n; ++j) {
            if (x[j][i] != 0) {
                pivot = j;
                break;
            }
        }
```

```
        if (x[pivot][i] == 0) {
            return 0;
        }
        if (pivot != i) {
            std::swap(x[i], x[pivot]);
            res = (MOD - res) % MOD;
        }
        for (i32 j = i + 1; j < x.n; ++j) {
            if (x[j][i] == 0) continue;
            while (x[i][i] != 0) {
                i64 fix = x[j][i] / x[i][i];
                for (i32 k = i; k < x.n; ++k) {
                    x[j][k] = ((x[j][k] - fix * x[i][k]) % MOD + MOD) % MOD;
                }
                for (i32 k = i; k < x.n; ++k) {
                    std::swap(x[i][k], x[j][k]);
                }
                res = (MOD - res) % MOD;
            }
            for (i32 k = i; k < x.n; ++k) {
                std::swap(x[i][k], x[j][k]);
            }
            res = (MOD - res) % MOD;
        }
        res = res * x[i][i] % MOD;
    }
    return res;
}
```

3.5 Power-Inv

```
const i64 MOD = 1E9 + 7;

i64 power(i64 a, i64 b, i64 p = MOD) {
    i64 res = 1;
    while (b) {
        if (b & 1) res = res * a % p;
        a = a * a % p;
        b >>= 1;
    }
    return res;
}

i64 inv(i64 a, i64 p = MOD) {
    return power(a, p - 2, p);
}

i64 pow(i64 a, i64 b, i64 p = MOD) {
    if (b < 0) {
        return power(inv(a, p), std::abs(b), p);
    } else {
        return power(a, b, p);
    }
}
```

3.6 Xor

```
i64 rangeXor(i64 a, i64 b) {
    std::function<i64(i64)> pref = [](i64 x) {
        if (x % 4 == 0) {
            return x;
        } else if (x % 4 == 1) {
            return 1LL;
        } else if (x % 4 == 2) {
            return x + 1;
        } else {
            return 0LL;
        }
    };
    return pref(b) ^ pref(a - 1);
};
```

4 数据结构

4.1 01-Trie

```
const i32 MAXH = 30;

struct Trie {
    i32 n, tot, root;
    std::vector<i32> w, sub;
    std::vector<std::array<i32, 2>> chd;
    Trie(i32 n) : n(n), tot(0), root(0) {
        i32 siz = n * (i32)std::ceil(std::__lg(n));
        w.assign(siz, 0);
        sub.assign(siz, 0);
        chd.assign(siz, {0, 0});
    }
    void maintain(i32 x) {
        w[x] = sub[x] = 0;
        if (chd[x][0]) {
            w[x] += w[chd[x][0]];
            sub[x] ^= sub[chd[x][0]] << 1;
        }
```

```
        if (chd[x][1]) {
            w[x] += w[chd[x][1]];
            sub[x] ^= (sub[chd[x][1]] << 1) | (w[chd[x][1]] & 1);
        }
        w[x] = w[x] & 1;
    }
    void insert(i32 &p, i32 x, i32 dep) {
        if (!p) {
            p = ++tot;
        }
        if (dep >= MAXH) {
            w[p]++;
            return;
        }
        insert(chd[p][x & 1], x >> 1, dep + 1);
        maintain(p);
    }
    void insert(i32 x) {
        insert(root, x, 0);
    }
    void erase(i32 &p, i32 x, i32 dep) {
        if (dep >= MAXH) {
            w[p]--;
            return;
        }
        erase(chd[p][x & 1], x >> 1, dep + 1);
        maintain(p);
    }
    void erase(i32 x) {
        erase(root, x, 0);
    }
    void addAll(i32 x = 0) {
        std::swap(chd[x][0], chd[x][1]);
        if (chd[x][0]) addAll(chd[x][0]);
        maintain(x);
    }
    i32 merge(Trie u, i32 a = 0, i32 b = 0) {
        if (!a) return b;
        if (!b) return a;
        w[a] = w[a] + u.w[b];
        sub[a] = sub[a] ^ u.sub[b];
        chd[a][0] = merge(u, chd[a][0], u.chd[b][0]);
        chd[a][1] = merge(u, chd[a][1], u.chd[b][1]);
        return a;
    }
};
```

4.2 LeftistTree

```
template <typename Info, typename Tag, typename Func = std::less<Info>>
struct LeftistTree {
    struct Node {
        Info info;
        Tag tag;
        i32 dis = 0;
        Node *lc = nullptr;
        Node *rc = nullptr;
        Node(const Info &x) : info(x) {}
    } *root = nullptr;
    i32 siz = 0;
    Func cmp;
    LeftistTree() = default;
    LeftistTree(Func cmp) : cmp(cmp) {}
    ~LeftistTree() {
        clear();
    }
    void abdicate() {
        root = nullptr;
        siz = 0;
    }
    i32 dist(Node *x) {
        if (x == nullptr) {
            return -1;
        }
        return x->dis;
    }
    void pushdown(Node *x) {
        if (x == nullptr) {
            return;
        }
        x->info.apply(x->tag);
        if (x->lc != nullptr) {
            x->lc->tag.apply(x->tag);
        }
        if (x->rc != nullptr) {
            x->rc->tag.apply(x->tag);
        }
        x->tag = Tag{};
    }
    Node* merge(Node* x, Node* y) {
        if (x == nullptr) return y;
        if (y == nullptr) return x;
        pushdown(x);
        pushdown(y);
        if (cmp(x->info, y->info)) {
```

```
            std::swap(x, y);
        }
        x->rc = merge(x->rc, y);
        if (dist(x->lc) < dist(x->rc)) {
            std::swap(x->lc, x->rc);
        }
        x->dis = dist(x->rc) + 1;
        return x;
    }
    void merge(LeftistTree &x) {
        root = merge(root, x.root);
        siz += x.size();
        x.abdicate();
    }
    void clear() {
        if (root == nullptr) return;
        std::queue<Node*> q;
        q.push(root);
        while (!q.empty()) {
            Node *u = q.front();
            q.pop();
            if (u->lc != nullptr) {
                q.push(u->lc);
            }
            if (u->rc != nullptr) {
                q.push(u->rc);
            }
            delete u;
        }
        abdicate();
    }
    void apply(const Tag &v) {
        if (root != nullptr) {
            root->tag.apply(v);
        }
    }
    void push(Info x) {
        Node *temp = new Node(x);
        root = merge(root, temp);
        siz++;
    }
    void pop() {
        pushdown(root);
        Node *temp = root;
        root = merge(root->lc, root->rc);
        delete temp;
        siz--;
    }
    bool empty() {
        return root == nullptr;
    }
    Info top() {
        pushdown(root);
        return root->info;
    }
    i32 size() {
        return siz;
    }
};

struct Tag {
    i64 add = 0;
    void apply(const Tag &v) {
        add += v.add;
    }
};

struct Info {
    i64 val = 0;
    void apply(const Tag &v) {
        val += v.add;
    }
    bool operator < (const Info &u) const {
        return val < u.val;
    }
    bool operator > (const Info &u) const {
        return val > u.val;
    }
};
```

4.3 SparseTableByEnar

```
template <typename T, typename Func = std::function<T(const T&, const T&)>>
struct ST {
    ST(const std::vector<T> &v, Func func =
        [](const T& a, const T& b) {
            return std::max(a, b);
        }
    ) : func(std::move(func)) {
        int k = std::__lg(v.size());
        st = std::vector<std::vector<T>>(k + 1, std::vector<T>(v.size()));
        st[0] = v;
        for(int i = 0; i < k; ++i) {
            for(int j = 0; j + (1 << (i + 1)) - 1 < v.size(); ++j) {
                st[i + 1][j] = this->func(st[i][j], st[i][j + (1 << i)]);
            }
```

```
        }
    }
    T range(int l, int r) {
        int t = std::__lg(r - l + 1);
        return func(st[t][l], st[t][r + 1 - (1 << t)]);
    }
    std::vector<std::vector<T>> st;
    Func func;
};
```

4.4 并查集

4.4.1 DSU-Rollback

```
struct DSU {
    std::vector<i32> p, siz;
    std::vector<std::array<i32, 2>> his;
    DSU(i32 n) : siz(n + 1, 1), p(n + 1) {
        std::iota(p.begin(), p.end(), 0);
    }
    i32 find(i32 x) {
        while (p[x] != x) {
            x = p[x];
        }
        return x;
    }
    bool merge(i32 x, i32 y) {
        x = find(x);
        y = find(y);
        if (x == y) {
            return false;
        }
        if (siz[x] < siz[y]) {
            std::swap(x, y);
        }
        his.push_back({x, y});
        siz[x] += siz[y];
        p[y] = x;
        return true;
    }
    i32 time() {
        return his.size();
    }
    void revert(i32 tm) {
        while (his.size() > tm) {
            auto [x, y] = his.back();
            his.pop_back();
            p[y] = y;
            siz[x] -= siz[y];
        }
    }
};
```

4.4.2 DSU

```
struct DSU {
    std::vector<i32> p, siz;
    DSU() {}
    DSU(i32 n) {
        init(n);
    }
    void init(i32 n) {
        p.resize(n);
        siz.assign(n, 1);
        std::iota(p.begin(), p.end(), 0);
    }
    i32 find(i32 x) {
        while (x != p[x]) x = p[x] = p[p[x]];
        return x;
    }
    bool same(i32 x, i32 y) {
        return find(x) == find(y);
    }
    bool merge(i32 x, i32 y) {
        x = find(x);
        y = find(y);
        if (x == y) {
            return false;
        } else {
            p[y] = x;
            siz[x] += siz[y];
            return true;
        }
    }
    i32 size(i32 x) {
        return siz[find(x)];
    }
};
```

4.5 树状数组

4.5.1 Fenwick-Range

```
template<typename T>
struct Fenwick {
    i32 n;
```

```
    std::vector<T> s, t;
    Fenwick(i32 n) : n(n) {
        s.assign(n, T{});
        t.assign(n, T{});
    }
    void baseApply(i32 x, const T &v) {
        for (i32 i = x + 1; i <= n; i += i & -i) {
            s[i - 1] = s[i - 1] + v;
            t[i - 1] = t[i - 1] + x * v;
        }
    }
    void rangeApply(i32 l, i32 r, const T &v) {
        baseApply(l, v);
        baseApply(r + 1, -v);
    }
    void apply(i32 x, const T &v) {
        rangeApply(x, x, v);
    }
    T baseQuery(i32 x) {
        T res{};
        for (i32 i = x; i > 0; i -= i & -i) {
            res = res + x * s[i - 1] - t[i - 1];
        }
        return res;
    }
    T rangeQuery(i32 l, i32 r) {
        return baseQuery(r) - baseQuery(l - 1);
    }
    T query(i32 x) {
        return rangeQuery(x, x);
    }
};
```

4.5.2 Fenwick

```
template<typename T>
struct Fenwick {
    i32 n;
    std::vector<T> tree;
    Fenwick(i32 n = 0) {
        init(n);
    }
    void init(i32 n) {
        this->n = n;
        tree.assign(n, T{});
    }
    void apply(i32 x, const T &v) {
        for (i32 i = x + 1; i <= n; i += i & -i) {
            tree[i - 1] = tree[i - 1] + v;
        }
    }
    T query(i32 x) {
        T res{};
        for (i32 i = x; i > 0; i -= i & -i) {
            res = res + tree[i - 1];
        }
        return res;
    }
    T rangeQuery(i32 l, i32 r) {
        return query(r) - query(l);
    }
};
```

4.6 线段树

4.6.1 LazySegmentTree-Extra

```
const i64 INF = 1E18;
const i64 MOD = 1E9 + 7;

template <typename Info, typename Tag>
struct SegmentTree {
    i32 n;
    std::vector<Tag> tag;
    std::vector<Info> info;
    SegmentTree(i32 n) : n(n), info(4 << std::__lg(n)), tag(4 << std::__lg(n))
    {}
    SegmentTree(const std::vector<auto> &v) : SegmentTree(v.size()) {
        auto build = [&](auto self, i32 p, i32 l, i32 r) {
            if(r == l) {
                info[p].init(v[l]);
                return;
            }
            i32 m = (l + r) / 2;
            self(self, p << 1, l, m);
            self(self, p << 1 | 1, m + 1, r);
            pull(p);
        };
        build(build, 1, 0, n - 1);
    }
    void add(i32 p, const Tag &v) {
        info[p].apply(v);
        tag[p].apply(v);
    }
    void mul(i32 p, const Tag &v) {
        info[p].multi(v);
```

```
        tag[p].multi(v);
    }
    void update(i32 p, const Tag &v) {
        info[p].update(v);
        tag[p].update(v);
    }
    void pull(i32 p) {
        info[p] = info[p << 1] + info[p << 1 | 1];
    }
    void push(i32 p) {
        update(p << 1, tag[p]);
        update(p << 1 | 1, tag[p]);
        tag[p] = Tag();
    }
    void rangeApply(i32 p, i32 x, i32 y, i32 l, i32 r, const Tag &v) {
        if (x <= l && y >= r) {
            add(p, v);
            return;
        }
        push(p);
        i32 m = (l + r) / 2;
        if (x <= m) rangeApply(p << 1, x, y, l, m, v);
        if (y > m) rangeApply(p << 1 | 1, x, y, m + 1, r, v);
        pull(p);
    }
    void rangeApply(i32 x, i32 y, const Tag &v) {
        rangeApply(1, x, y, 0, n - 1, v);
    }
    void apply(i32 x, const Tag &v) {
        rangeApply(1, x, x, 0, n - 1, v);
    }
    void rangeMultiply(i32 p, i32 x, i32 y, i32 l, i32 r, const Tag &v) {
        if (x <= l && y >= r) {
            mul(p, v);
            return;
        }
        push(p);
        i32 m = (l + r) / 2;
        if (x <= m) rangeMultiply(p << 1, x, y, l, m, v);
        if (y > m) rangeMultiply(p << 1 | 1, x, y, m + 1, r, v);
        pull(p);
    }
    void rangeMultiply(i32 x, i32 y, const Tag &v) {
        rangeMultiply(1, x, y, 0, n - 1, v);
    }
    void multiply(i32 x, const Tag &v) {
        rangeMultiply(1, x, x, 0, n - 1, v);
    }
    Info rangeQuery(i32 p, i32 x, i32 y, i32 l, i32 r) {
        if (x <= l && y >= r) return info[p];
        Info res;
        push(p);
        i32 m = (l + r) / 2;
        if (x <= m) res = res + rangeQuery(p << 1, x, y, l, m);
        if (y > m) res = res + rangeQuery(p << 1 | 1, x, y, m + 1, r);
        pull(p);
        return res;
    }
    Info rangeQuery(i32 x, i32 y) {
        return rangeQuery(1, x, y, 0, n - 1);
    }
    Info query(i32 x) {
        return rangeQuery(1, x, x, 0, n - 1);
    }
};

struct Tag {
    i64 add = 0;
    i64 mul = 1;
    void apply(const Tag &v) {
        add = (add + v.add) % MOD;
    }
    void multi(const Tag &v) {
        mul = mul * v.mul % MOD;
        add = add * v.mul % MOD;
    }
    void update(const Tag &v) {
        multi(v);
        apply(v);
    }
};

struct Info {
    i64 sum = 0;
    i64 len = 1;
    void init(const i64 &x) {
        sum = x;
    }
    void apply(const Tag &v) {
        sum = (sum + len * v.add % MOD) % MOD;
    }
    void multi(const Tag &v) {
        sum = sum * v.mul % MOD;
    }
    void update(const Tag &v) {
        multi(v);
```

```
        apply(v);
    }
    Info operator + (const Info &a) {
        Info res;
        res.sum = (sum + a.sum) % MOD;
        res.len = (len + a.len) % MOD;
        return res;
    }
};
```

4.6.2 LazySegmentTree-Short

```
template <typename T>
struct LazySegmentTree {
    i32 n;
    std::vector<T> tree, tag;
    LazySegmentTree(i32 n) : n(n), tree(4 << std::__lg(n)), tag(4 << std::__lg(
    n)) {}
    LazySegmentTree(std::vector<T> &v) : LazySegmentTree(v.size()) {
        auto build = [&](auto &&build, i32 p, i32 l, i32 r) {
            if(r == l) {
                tree[p] = v[l];
                return;
            }
            i32 m = (l + r) / 2;
            build(p << 1, l, m);
            build(p << 1 | 1, m + 1, r);
            pull(p);
        };
        build(1, 0, n - 1);
    }
    void pull(i32 p) {
        tree[p] = tree[p << 1] + tree[p << 1 | 1];
    }
    void push(i32 p, i32 l, i32 r) {
        if (l != r && tag[p] > 0) {
            i32 m = (l + r) / 2;
            tag[p << 1] += tag[p];
            tag[p << 1 | 1] += tag[p];
            tree[p << 1] += tag[p] * (m - l + 1);
            tree[p << 1 | 1] += tag[p] * (r - m);
            tag[p] = 0;
        }
    }
    void rangeApply(i32 p, i32 x, i32 y, i32 l, i32 r, const T &v) {
        if (x <= l && y >= r) {
            tree[p] += (r - l + 1) * v;
            tag[p] += v;
            return;
        }
        push(p, l, r);
        i32 m = (l + r) / 2;
        if (x <= m) rangeApply(p << 1, x, y, l, m, v);
        if (y > m) rangeApply(p << 1 | 1, x, y, m + 1, r, v);
        pull(p);
    }
    void rangeApply(i32 x, i32 y, const T &v) {
        rangeApply(1, x, y, 0, n - 1, v);
    }
    void apply(i32 x, const T &v) {
        rangeApply(1, x, x, 0, n - 1, v);
    }
    T rangeQuery(i32 p, i32 x, i32 y, i32 l, i32 r) {
        if (x <= l && y >= r) return tree[p];
        T res = 0;
        push(p, l, r);
        i32 m = (l + r) / 2;
        if (x <= m) res += rangeQuery(p << 1, x, y, l, m);
        if (y > m) res += rangeQuery(p << 1 | 1, x, y, m + 1, r);
        pull(p);
        return res;
    }
    T rangeQuery(i32 x, i32 y) {
        return rangeQuery(1, x, y, 0, n - 1);
    }
    T query(i32 x) {
        return rangeQuery(1, x, x, 0, n - 1);
    }
};
```

4.6.3 LazySegmentTree

```
const i64 INF = 1E18;

template<typename Info, typename Tag>
struct SegmentTree {
    i32 n;
    std::vector<Tag> tag;
    std::vector<Info> info;
    SegmentTree(i32 n) : n(n), info(4 << std::__lg(n)), tag(4 << std::__lg(n))
    {}
    SegmentTree(const std::vector<auto> &v) : SegmentTree(v.size()) {
        auto build = [&](auto self, i32 p, i32 l, i32 r) {
            if(r == l) {
                info[p].init(v[l]);
                return;
```

```
            }
            i32 m = (l + r) / 2;
            self(self, p << 1, l, m);
            self(self, p << 1 | 1, m + 1, r);
            pull(p);
        };
        build(build, 1, 0, n - 1);
    }
    void update(i32 p, const Tag &v) {
        info[p].apply(v);
        tag[p].apply(v);
    }
    void pull(i32 p) {
        info[p] = info[p << 1] + info[p << 1 | 1];
    }
    void push(i32 p, i32 l, i32 r) {
        update(p << 1, tag[p]);
        update(p << 1 | 1, tag[p]);
        tag[p] = Tag();
    }
    void rangeApply(i32 p, i32 x, i32 y, i32 l, i32 r, const Tag &v) {
        if (x <= l && y >= r) {
            update(p, v);
            return;
        }
        push(p, l, r);
        i32 m = (l + r) / 2;
        if (x <= m) rangeApply(p << 1, x, y, l, m, v);
        if (y > m) rangeApply(p << 1 | 1, x, y, m + 1, r, v);
        pull(p);
    }
    void rangeApply(i32 x, i32 y, const Tag &v) {
        rangeApply(1, x, y, 0, n - 1, v);
    }
    void apply(i32 x, const Tag &v) {
        rangeApply(1, x, x, 0, n - 1, v);
    }
    Info rangeQuery(i32 p, i32 x, i32 y, i32 l, i32 r) {
        if (x <= l && y >= r) return info[p];
        Info res;
        push(p, l, r);
        i32 m = (l + r) / 2;
        if (x <= m) res = res + rangeQuery(p << 1, x, y, l, m);
        if (y > m) res = res + rangeQuery(p << 1 | 1, x, y, m + 1, r);
        pull(p);
        return res;
    }
    Info rangeQuery(i32 x, i32 y) {
        return rangeQuery(1, x, y, 0, n - 1);
    }
    Info query(i32 x) {
        return rangeQuery(1, x, x, 0, n - 1);
    }
};

struct Tag {
    i64 add = 0;
    void apply(const Tag &v) {
        add += v.add;
    }
};

struct Info {
    i64 sum = 0, len = 1;
    i64 mn = INF, mx = -INF;
    void init(const i64 &x) {
        mn = mx = x;
        sum = x;
    }
    void apply(const Tag &v) {
        sum += len * v.add;
        mn += v.add;
        mx += v.add;
    }
    Info operator + (const Info &a) {
        Info res;
        res.mn = std::min(mn, a.mn);
        res.mx = std::max(mx, a.mx);
        res.sum = sum + a.sum;
        res.len = len + a.len;
        return res;
    }
};
```

4.6.4 PersistentSegmentTreeByEnar

```
template<typename Info, typename Tag>
struct PersistentTree {
    struct Node {
        i32 l = 0, r = 0;
        Info info;
        Tag tag;
    };
#define ls(x) (node[x].l)
#define rs(x) (node[x].r)
    PersistentTree(i32 n) : PersistentTree(std::vector<Info>(n + 1)) {}
```

```
    PersistentTree(const std::vector<Info> &init) : n((i32)init.size() - 1) {
        node.reserve(n << 3);
        auto build = [&](auto self, i32 l, i32 r) ->i32 {
            node.push_back(Node());
            i32 id = node.size() - 1;
            if(l == r) {
                node[id].info = init[l];
            } else {
                i32 mid = (l + r) / 2;
                ls(id) = self(self, l, mid);
                rs(id) = self(self, mid + 1, r);
                node[id].info = node[ls(id)].info + node[rs(id)].info;
            }
            return id;
        };
        root.push_back(build(build, 1, n));
    };
    i32 update(i32 ver, i32 pos, const Info &val) {
        root.push_back(update(root[ver], 1, n, pos, val));
        return root.size() - 1;
    }
    i32 update(i32 ver, i32 pos, const Tag &dx) {
        root.push_back(update(root[ver], 1, n, pos, dx));
        return root.size() - 1;
    }
    Info query(i32 ver, i32 pos) {
        return rangeQuery(ver, pos, pos);
    }
    Info rangeQuery(i32 ver, i32 l, i32 r) {
        return rangeQuery(root[ver], 1, n, l, r);
    }
    i32 update(i32 lst, i32 l, i32 r, const i32 &pos, const Info &val) {
        node.push_back(node[lst]);
        i32 id = node.size() - 1;
        if(l == r) {
            node[id].info = val;
        } else {
            i32 mid = (l + r) / 2;
            if(pos <= mid) {
                ls(id) = update(ls(lst), l, mid, pos, val);
            } else if(pos > mid) {
                rs(id) = update(rs(lst), mid + 1, r, pos, val);
            }
            node[id].info = node[ls(id)].info + node[rs(id)].info;
        }
        return id;
    }
    i32 update(i32 lst, i32 l, i32 r, const i32 &pos, const Tag &dx) {
        node.push_back(node[lst]);
        i32 id = node.size() - 1;
        if(l == r) {
            node[id].info.apply(dx);
        } else {
            i32 mid = (l + r) / 2;
            if(pos <= mid) {
                ls(id) = update(ls(lst), l, mid, pos, dx);
            } else if(pos > mid) {
                rs(id) = update(rs(lst), mid + 1, r, pos, dx);
            }
            node[id].info = node[ls(id)].info + node[rs(id)].info;
        }
        return id;
    }
    Info rangeQuery(i32 id, i32 l, i32 r, const i32 &x, const i32 &y) {
        if(x <= l && r <= y) {
            return node[id].info;
        }
        i32 mid = (l + r) / 2;
        Info res;
        if(x <= mid) {
            res = res + rangeQuery(ls(id), l, mid, x, y);
        }
        if(y > mid) {
            res = res + rangeQuery(rs(id), mid + 1, r, x, y);
        }
        return res;
    }
    i32 kth(i32 verl, i32 verr, i32 k) {
        return kth(root[verl], root[verr], 1, n, k);
    }
    i32 kth(i32 idx, i32 idy, i32 l, i32 r, i32 k) { //静态区间第k小，不支持修改
        if(l >= r) return l;
        i32 mid = (l + r) / 2;
        i32 dx = node[ls(idy)].info.sum - node[ls(idx)].info.sum;
        if(dx >= k) {
            return kth(ls(idx), ls(idy), l, mid, k);
        } else {
            return kth(rs(idx), rs(idy), mid + 1, r, k - dx);
        }
    }
#undef ls
#undef rs
    const i32 n;
    std::vector<Node> node;
    std::vector<i32> root;
```

```
};

struct Tag {
    Tag(i32 dx = 0) : add(dx) {}
    i32 add = 0;
    void apply(const Tag &dx) {
        add += dx.add;
    }
};

struct Info {
    i32 sum = 0;
    void apply(const Tag &dx) {
        sum += dx.add;
    }
};

Info operator+(const Info &x, const Info &y) {
    Info res;
    res.sum = x.sum + y.sum;
    return res;
}
```

5 数论

5.1 CRT

```
i64 exgcd(i64 a, i64 b, i64 &x, i64 &y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    i64 d = exgcd(b, a % b, y, x);
    y -= (a / b) * x;
    return d;
}

i64 crt(i32 k, std::vector<i64> a, std::vector<i64> r) {
    i64 n = 1;
    i64 ans = 0;
    for (i32 i = 0; i < k; ++i) {
        n = n * r[i];
    }
    for (i32 i = 0; i < k; ++i) {
        i64 m = n / r[i];
        i64 b, y;
        exgcd(m, r[i], b, y);
        ans = (ans + a[i] * m * b % n) % n;
    }
    return (ans % n + n) % n;
}
```

5.2 Euclidean-Like

```
const i32 I2 = 499122177, I6 = 166374059;

struct Euclidean {
    i64 f, g, h;
    Euclidean() : f(0), g(0), h(0) {}
    Euclidean(i64 n_, i64 a_, i64 b_, i64 c_, i64 p_) {
        Euclidean tmp = euc(n_, a_, b_, c_, p_);
        *this = tmp;
    }
    Euclidean euc(i64 n, i64 a, i64 b, i64 c, i64 p) {
        i64 ac = a / c;
        i64 bc = b / c;
        i64 m = (a * n + b) / c;
        i64 n1 = n + 1;
        i64 n21 = n * 2 + 1;
        Euclidean d;
        if (a == 0) {
            d.f = bc * n1 % p;
            d.g = bc * n % p * n1 % p * I2 % p;
            d.h = bc * bc % p * n1 % p;
            return d;
        }
        if (a >= c || b >= c) {
            d.f = n * n1 % p * I2 % p * ac % p
                + bc * n1 % p;
            d.g = ac * n % p * n1 % p * n21 % p * I6 % p
                + bc * n % p * n1 % p * I2 % p;
            d.h = ac * ac % p * n % p * n1 % p * n21 % p * I6 % p
                + ac * bc % p * n % p * n1 % p
                + bc * bc % p * n1 % p;
            d.f %= p;
            d.g %= p;
            d.h %= p;
            Euclidean e = euc(n, a % c, b % c, c, p);
            d.h += e.h + 2 * bc % p * e.f % p + 2 * ac % p * e.g % p;
            d.g += e.g;
            d.f += e.f;
            d.f %= p;
            d.g %= p;
            d.h %= p;
```

```
            return d;
        }
        Euclidean e = euc(m - 1, c, c - b - 1, a, p);
        d.f = n * m % p - e.f;
        d.f = (d.f % p + p) % p;
        d.g = m * n % p * n1 % p - e.h - e.f;
        d.g = (d.g * I2 % p + p) % p;
        d.h = n * m % p * (m + 1) % p - 2 * e.g - 2 * e.f - d.f;
        d.h = (d.h % p + p) % p;
        return d;
    }
};
```

5.3 Ex-GCD

```
i64 exgcd(i64 a, i64 b, i64 &x, i64 &y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    i64 d = exgcd(b, a % b, y, x);
    y -= (a / b) * x;
    return d;
}
```

5.4 Factorize

```
const i64 BASE1[7] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022};
const i64 BASE2[9] = {2, 3, 5, 7, 11, 13, 17, 19, 23};

i64 power(i64 a, i64 b, i64 p) {
    i64 res = 1;
    while (b) {
        if (b & 1) res = (i128)res * a % p;
        a = (i128)a * a % p;
        b >>= 1;
    }
    return res;
}

bool millerRabin(i64 n) {
    if (n < 3 || n % 2 == 0) return n == 2;
    if (n % 3 == 0) return n == 3;
    i64 u = n - 1, t = 0;
    while (u % 2 == 0) {
        u /= 2;
        t++;
    }
    for (i32 i = 0; i < 9; ++i) {
        i64 v = power(BASE1[i], u, n);
        if (v == 1) continue;
        for (i32 s = 0; s <= t; ++s) {
            if(s == t) return false;
            if(v == n - 1) break;
            v = (i128)v * v % n;
        }
    }
    return true;
}

i64 pollardRho(i64 n) {
    static std::mt19937_64 rng(std::chrono::steady_clock::now().
    time_since_epoch().count());
    std::uniform_int_distribution<i64> rangeRand(1, n - 1);
    i64 c = rangeRand(rng);
    auto f = [&](i64 x) -> i64 {
        return (static_cast<i128>(x) * x + c) % n;
    };
    i64 t = f(0), r = f(t);
    while (t != r) {
        i64 d = std::__gcd(std::abs(t - r), n);
        if (d > 1) return d;
        r = f(f(r));
        t = f(t);
    }
    return n;
}

std::vector<i64> factorize(i64 n) {
    std::vector<i64> p;
    std::function<void(i64)> work = [&](i64 num) {
        if (num <= 10000) {
            for (i32 i = 2; i * i <= num; ++i) {
                while (num % i == 0) {
                    p.push_back(i);
                    num /= i;
                }
            }
            if (num > 1) p.push_back(num);
            return;
        }
        if (millerRabin(num)) {
            p.push_back(num);
            return;
```

```
        }
        i64 x = num;
        while (x == num) x = pollardRho(num);
        work(num / x);
        work(x);
    };
    work(n);
    std::sort(p.begin(), p.end());
    return p;
}
```

5.5 GCD-LCM

```
i64 gcd(i64 a, i64 b) {
   return a ? gcd(b % a, a) : b;
}

i64 lcm(i64 a, i64 b) {
   return a * b / gcd(a, b);
}
```

5.6 LCE

```
i64 exgcd(i64 a, i64 b, i64 &x, i64 &y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    i64 d = exgcd(b, a % b, y, x);
    y -= (a / b) * x;
    return d;
}

bool lieu(i64 a, i64 b, i64 c, i64 &x, i64 &y) {
    i64 d = exgcd(a, b, x, y);
    if (c % d == 0) return false;
    i64 k = c / d;
    x *= k;
    y *= k;
    return true;
}
```

5.7 Miller-Rabin

```
const i64 BASE1[7] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022};
const i64 BASE2[9] = {2, 3, 5, 7, 11, 13, 17, 19, 23};

i64 power(i64 a, i64 b, i64 p) {
    i64 res = 1;
    while (b) {
        if (b & 1) res = (i128)res * a % p;
        a = (i128)a * a % p;
        b >>= 1;
    }
    return res;
}

bool millerRabin(i64 n) {
    if (n < 3 || n % 2 == 0) return n == 2;
    if (n % 3 == 0) return n == 3;
    i64 u = n - 1, t = 0;
    while (u % 2 == 0) {
        u /= 2;
        t++;
    }
    for (i32 i = 0; i < 9; ++i) {
        i64 v = power(BASE1[i], u, n);
        if (v == 1) continue;
        for (i32 s = 0; s <= t; ++s) {
            if(s == t) return false;
            if(v == n - 1) break;
            v = (i128)v * v % n;
        }
    }
    return true;
}
```

5.8 Phi

```
std::vector<bool> vis;
std::vector<i32> phi, primes;

void getPhi(i32 n) {
    phi.assign(n + 1, 0);
    vis.assign(n + 1, false);
    phi[1] = 1;
    for (i32 i = 2; i <= n; ++i) {
        if (!vis[i]) {
            primes.push_back(i);
            phi[i] = i - 1;
        }
        for (const auto &p : primes) {
            i64 m = i * p;
```

```
            if (m > n) {
                break;
            }
            vis[m] = true;
            if (i % p == 0) {
                phi[m] = phi[i] * p;
                break;
            } else {
                phi[m] = phi[i] * (p - 1);
            }
        }
    }
}
```

5.9 Pollard-Rho

```
std::mt19937_64 rnd(std::chrono::steady_clock::now().time_since_epoch().count()
    );

i64 pollardRho(i64 n) {
    std::uniform_int_distribution<i64> rangeRand(1, n - 1);
    i64 c = rangeRand(rnd);
    auto f = [&](i64 x) -> i64 {
        return (static_cast<i128>(x) * x + c) % n;
    };
    i64 t = f(0), r = f(t);
    while (t != r) {
        i64 d = std::gcd(std::abs(t - r), n);
        if (d > 1) return d;
        r = f(f(r));
        t = f(t);
    }
    return n;
}
```

5.10 Sieve

```
std::vector<i32> minp, primes;

void sieve(i32 n) {
    primes.clear();
    minp.assign(n + 1, 0);
    for(i32 i = 2; i <= n; i++) {
        if (minp[i] == 0) {
            minp[i] = i;
            primes.push_back(i);
        }
        for (auto p : primes) {
            if (i * p > n) {
                break;
            }
            minp[i * p] = p;
            if (p == minp[i]) {
                break;
            }
        }
    }
}
```

6 杂类

6.1 QuickRead

```
inline i64 read() {
    char s;
    i64 k = 0, base = 1;
    while ((s = getchar()) != '-' && s != EOF && !(s >= '0' && s <= '9'));
    if (s == EOF) exit(0);
    if (s == '-') base = -1, s = getchar();
    while (s >= '0' && s <= '9') {
        k = k * 10 + (s - '0');
        s = getchar();
    }
    return k * base;
}

inline void write(i64 x) {
    if (x > 9) write(x / 10);
    putchar(x % 10 | 48);
}
```

7 计算几何

7.1 BinaryHull

```
const double EPS = 1E-10;

i32 fcmp(double x) {
    if (fabs(x) < EPS) return 0;
    else if (x < 0) return -1;
    else return 1;
}

```

```cpp
struct Vecteur {
    double x, y;
    Vecteur() {}
    Vecteur(double x, double y) : x(x), y(y) {}
    Vecteur operator + (const Vecteur &u) {
        return Vecteur(x + u.x, y + u.y);
    }
    Vecteur operator - (const Vecteur &u) {
        return Vecteur(x - u.x, y - u.y);
    }
    Vecteur operator * (const double &k) {
        return Vecteur(k * x, k * y);
    }
    Vecteur operator / (double &k) {
        if (k < EPS) k += EPS;
        return Vecteur(x / k, y / k);
    }
    bool operator < (const Vecteur &u) const {
        return (fcmp(x - u.x) == -1) || (fcmp(x - u.x) == 0 && fcmp(y - u.y) == -1);
    }
};
double dot(Vecteur a, Vecteur b) {
    return a.x * b.x + a.y * b.y;
}
double cross(Vecteur a, Vecteur b) {
    return a.x * b.y - a.y * b.x;
}
using Point = Vecteur;

auto binaryHull(std::vector<Point> p) {
    std::sort(p.begin(), p.end());
    std::vector<Point> up, dn;
    for (auto &u : p) {
        while (up.size() > 1 && cross(up.back() - up[up.size() - 2], u - up.back()) >= 0) {
            up.pop_back();
        }
        while (!up.empty() && up.back().x == u.x) {
            up.pop_back();
        }
        up.push_back(u);
        while (dn.size() > 1 && cross(dn.back() - dn[dn.size() - 2], u - dn.back()) <= 0) {
            dn.pop_back();
        }
        if (dn.empty() || dn.back().x < u.x) {
            dn.push_back(u);
        }
    }
    return std::make_pair(up, dn);
}
```

7.2 ConvexHull

```cpp
const double EPS = 1E-10;

i32 fcmp(double x) {
    if (fabs(x) < EPS) return 0;
    else if (x < 0) return -1;
    else return 1;
}

struct Vecteur {
    double x, y;
    Vecteur() {}
    Vecteur(double x, double y) : x(x), y(y) {}
    Vecteur operator + (const Vecteur &u) {
        return Vecteur(x + u.x, y + u.y);
    }
    Vecteur operator - (const Vecteur &u) {
        return Vecteur(x - u.x, y - u.y);
    }
    Vecteur operator * (const double &k) {
        return Vecteur(k * x, k * y);
    }
    Vecteur operator / (double &k) {
        if (k < EPS) k += EPS;
        return Vecteur(x / k, y / k);
    }
    bool operator < (const Vecteur &u) const {
        return (fcmp(x - u.x) == -1) || (fcmp(x - u.x) == 0 && fcmp(y - u.y) == -1);
    }
};
double dot(Vecteur a, Vecteur b) {
    return a.x * b.x + a.y * b.y;
}
double cross(Vecteur a, Vecteur b) {
    return a.x * b.y - a.y * b.x;
}
using Point = Vecteur;

auto convexHull(std::vector<Point> p) {
    std::sort(p.begin(), p.end());
    std::vector<Point> res;
```

```
    i32 n = p.size();
    for (i32 i = 0; i < n; ++i) {
        while (res.size() > 1 && cross(res.back() - res[res.size() - 2], p[i] - res.back()) <= 0) {
            res.pop_back();
        }
        res.push_back(p[i]);
    }
    i32 m = res.size();
    for (i32 i = n - 2; i >= 0; --i) {
        while (res.size() > m && cross(res.back() - res[res.size() - 2], p[i] - res.back()) <= 0) {
            res.pop_back();
        }
        res.push_back(p[i]);
    }
    if (res.size() > 1) res.pop_back();
    return res;
}
```

7.3 Line

```
const double EPS = 1E-10;

i32 fcmp(double x) {
    if (fabs(x) < EPS) return 0;
    else if (x < 0) return -1;
    else return 1;
}

struct Vecteur {
    double x, y;
    Vecteur() = default;
    Vecteur(double x, double y) : x(x), y(y) {}
    Vecteur(Vecteur a, Vecteur b) : x(b.x - a.x), y(b.y - a.y) {}
    Vecteur operator + (const Vecteur &u) const {
        return {x + u.x, y + u.y};
    }
    Vecteur operator - (const Vecteur &u) const {
        return {x - u.x, y - u.y};
    }
    Vecteur operator * (const double &k) const {
        return {k * x, k * y};
    }
    Vecteur operator / (double k) const {
        if (k < EPS) k += EPS;
        return {x / k, y / k};
    }
    bool operator < (const Vecteur &u) const {
        return (fcmp(x - u.x) == -1) || (fcmp(x - u.x) == 0 && fcmp(y - u.y) == -1);
    }
    bool operator == (const Vecteur &u) const {
        return (fcmp(x - u.x) == 0) && (fcmp(y - u.y) == 0);
    }
    double abs2() const {
        return x * x + y * y;
    }
    double abs() const {
        return std::sqrt(abs2());
    }
    double arg() const {
        return atan2(y, x);
    }
    Vecteur rotate(double rad) {
        return {x * cos(rad) - y * sin(rad), x * sin(rad) + y * cos(rad)};
    }
    Vecteur unit() {
        double len = abs();
        if (fcmp(len) == 0) {
            return Vecteur(0, 0);
        } else {
            return Vecteur(-y / len, x / len);
        }
    }
    Vecteur norm() {
        return {-y, x};
    }
};
double dot(Vecteur a, Vecteur b) {
    return a.x * b.x + a.y * b.y;
}
double cross(Vecteur a, Vecteur b) {
    return a.x * b.y - a.y * b.x;
}
using Point = Vecteur;

struct Line : Vecteur {
    Point p;
    Line() {}
    Line(Point p, Vecteur d) : Vecteur(d), p(p) {}
    Line(double k, double b) : Vecteur{1, k}, p{0, b} {}
    Line(double a, double b, double c) : Vecteur{-b, a}, p{0, -c / b} {}
    double operator () (const Vecteur &u) const {
        return dot(u, Vecteur(*this).norm()) + p.y * x;
    }
```

```
    bool operator < (const Line &u) const {
        i32 rst = fcmp(Vecteur(*this).arg() - u.arg());
        if (rst == 0) {
            return (u.p.x - p.x) * (u.p.y - p.x) > EPS;
        } else {
            return rst < 0;
        }
    }
};
Point junct(Line a, Line b) {
    Vecteur u = a.p - b.p;
    double t = cross(Vecteur(b), u) / cross(Vecteur(a), Vecteur(b));
    return a.p + Vecteur(a) * t;
}
Vecteur refl(Vecteur v, Line l) {
    return v - Vecteur(l) * (l(v) / l.abs2() * 2);
}
Vecteur proj(Vecteur v, Line l) {
    return v - Vecteur(l) * (l(v) / l.abs2());
}
double dist(Point p, Line l) {
    return l(p) / p.abs();
}
bool isPara(Line a, Line b) {
    return !fcmp(cross(a, b));
}
bool isVert(Line a, Line b) {
    return !fcmp(dot(a, b));
}
bool isSdrt(Line a, Line b) {
    return isPara(a, b) && dot(a, b) > 0;
}
bool online(Point p, Line l) {
    return !fcmp(l(p));
}
```

7.4 RotatingCalipers

```
const double EPS = 1E-10;

i32 fcmp(double x) {
    if (fabs(x) < EPS) return 0;
    else if (x < 0) return -1;
    else return 1;
}

struct Vecteur {
    double x, y;
    Vecteur() {}
    Vecteur(double x, double y) : x(x), y(y) {}
    Vecteur(Vecteur a, Vecteur b) : x(b.x - a.x), y(b.y - a.y) {}
    Vecteur operator + (const Vecteur &u) {
        return Vecteur(x + u.x, y + u.y);
    }
    Vecteur operator - (const Vecteur &u) {
        return Vecteur(x - u.x, y - u.y);
    }
    Vecteur operator * (const double &k) {
        return Vecteur(k * x, k * y);
    }
    Vecteur operator / (double &k) {
        if (k < EPS) k += EPS;
        return Vecteur(x / k, y / k);
    }
    bool operator < (const Vecteur &u) const {
        return (fcmp(x - u.x) == -1) || (fcmp(x - u.x) == 0 && fcmp(y - u.y) ==
     -1);
    }
    bool operator == (const Vecteur &u) const {
        return (fcmp(x - u.x) == 0) && (fcmp(y - u.y) == 0);
    }
    bool operator != (const Vecteur &u) const {
        return (fcmp(x - u.x) != 0) || (fcmp(y - u.y) != 0);
    }
};
double dot(Vecteur a, Vecteur b) {
    return a.x * b.x + a.y * b.y;
}
double cross(Vecteur a, Vecteur b) {
    return a.x * b.y - a.y * b.x;
}
double dist2(Vecteur a, Vecteur b) {
    return std::pow(a.x - b.x, 2) + std::pow(a.y - b.y, 2);
}
double dist(Vecteur a, Vecteur b) {
    return std::sqrt(dist2(a, b));
}
using Point = Vecteur;

auto convexHull(std::vector<Point> p) {
    std::sort(p.begin(), p.end());
    std::vector<Point> res;
    i32 n = p.size();
    for (i32 i = 0; i < n; ++i) {
        while (res.size() > 1 && cross(res.back() - res[res.size() - 2], p[i] -
     res.back()) <= 0) {
```

```
            res.pop_back();
        }
        res.push_back(p[i]);
    }
    i32 m = res.size();
    for (i32 i = n - 2; i >= 0; --i) {
        while (res.size() > m && cross(res.back() - res[res.size() - 2], p[i] - res.back()) <= 0) {
            res.pop_back();
        }
        res.push_back(p[i]);
    }
    if (res.size() > 1) res.pop_back();
    return res;
}

double rotateCalipers(std::vector<Point> h) {
    h.push_back(h.front());
    i32 n = h.size();
    if (n < 4) {
        return dist2(h[0], h[1]);
    }
    double res = 0;
    for (i32 i = 1, j = 1; i < n; ++i) {
        Vecteur b(h[i - 1], h[i]);
        while (cross(b, Vecteur(h[i - 1], h[j])) <= cross(b, Vecteur(h[i - 1], h[(j + 1) % n]))) {
            j = (j + 1) % n;
        }
        res = std::max({res, dist2(h[i - 1], h[j]), dist2(h[i], h[j])});
    }
    return res;
}
```

7.5 Segment

```
const double EPS = 1E-10;

i32 fcmp(double x) {
    if (fabs(x) < EPS) return 0;
    else if (x < 0) return -1;
    else return 1;
}

struct Vecteur {
    double x, y;
    Vecteur() {}
    Vecteur(double x, double y) : x(x), y(y) {}
    Vecteur operator + (Vecteur &u) {
        return Vecteur(x + u.x, y + u.y);
    }
    Vecteur operator - (Vecteur &u) {
        return Vecteur(x - u.x, y - u.y);
    }
    Vecteur operator * (double &k) {
        return Vecteur(k * x, k * y);
    }
    Vecteur operator / (double &k) {
        if (k < EPS) k += EPS;
        return Vecteur(x / k, y / k);
    }
    bool operator < (const Vecteur &u) const {
        return (fcmp(x - u.x) == -1) || (fcmp(x - u.x) == 0 && fcmp(y - u.y) == -1);
    }
    bool operator == (const Vecteur &u) const {
        return (fcmp(x - u.x) == 0) && (fcmp(y - u.y) == 0);
    }
    double arg() {
        return atan2(y, x);
    }
    double abs() {
        return std::sqrt(x * x + y * y);
    }
    Vecteur unit() {
        double len = abs();
        if (fcmp(len) == 0) return Vecteur(0, 0);
        return Vecteur(-y / len, x / len);
    }
    Vecteur rotate(double rad) {
        return Vecteur(x * cos(rad) - y * sin(rad), x * sin(rad) + y * cos(rad));
    }
};
double dot(Vecteur a, Vecteur b) {
    return a.x * b.x + a.y * b.y;
}
double cross(Vecteur a, Vecteur b) {
    return a.x * b.y - a.y * b.x;
}
double cosine(Vecteur a, Vecteur b) {
    return dot(a, b) / a.abs() / b.abs();
}
double arg(Vecteur a, Vecteur b) {
    return acos(cosine(a, b));
}
```

```
double area(Vecteur a, Vecteur b, Vecteur c) {
    return cross(b - a, c - a) / 2.0;
}
using Point = Vecteur;

i32 ccw(Point a, Point b, Point c) {
    i32 sign = fcmp((b - a) * (c - a));
    if (sign == 0) {
        if (fcmp((b - a) % (c - a)) == -1) return 2;
        if ((c - a).norm() > (b - a).norm() + EPS) return -2;
    }
    return sign;
}

struct Segment {
    Point x, y;
    Segment() {}
    Segment(Point x, Point y) : x(x), y(y) {}
    bool isCross(const Point &p) {
        return (p - x) % (p - y) < EPS && std::fabs((p - x) * (p - y)) < EPS;
    }
    bool isCross(const Segment &s) {
        return ccw(x.x, x.y, y.x) * ccw(x.x, x.y, y.y) <= 0
            && ccw(y.x, y.y, x.x) * ccw(y.x, y.y, x.y) <= 0;
    }
};
```

7.6 Vecteur

```
const double EPS = 1E-10;

i32 fcmp(double x) {
    if (fabs(x) < EPS) return 0;
    else if (x < 0) return -1;
    else return 1;
}

struct Vecteur {
    double x, y;
    Vecteur() {}
    Vecteur(double x, double y) : x(x), y(y) {}
    Vecteur(Vecteur a, Vecteur b) : x(b.x - a.x), y(b.y - a.y) {}
    Vecteur operator + (Vecteur &u) {
        return Vecteur(x + u.x, y + u.y);
    }
    Vecteur operator - (Vecteur &u) {
        return Vecteur(x - u.x, y - u.y);
    }
    Vecteur operator * (double &k) {
        return Vecteur(k * x, k * y);
    }
    Vecteur operator / (double &k) {
        if (k < EPS) k += EPS;
        return Vecteur(x / k, y / k);
    }
    bool operator < (const Vecteur &u) const {
        return (fcmp(x - u.x) == -1) || (fcmp(x - u.x) == 0 && fcmp(y - u.y) ==
     -1);
    }
    bool operator == (const Vecteur &u) const {
        return (fcmp(x - u.x) == 0) && (fcmp(y - u.y) == 0);
    }
    double arg() {
        return atan2(y, x);
    }
    double abs() {
        return std::sqrt(x * x + y * y);
    }
    Vecteur rotate(double rad) {
        return Vecteur(x * cos(rad) - y * sin(rad), x * sin(rad) + y * cos(rad)
     );
    }
    Vecteur unit() {
        double len = abs();
        if (fcmp(len) == 0) {
            return Vecteur(0, 0);
        } else {
            return Vecteur(-y / len, x / len);
        }
    }
    Vecteur norm() {
        return {-y, x};
    }
};
double dot(Vecteur a, Vecteur b) {
    return a.x * b.x + a.y * b.y;
}
double cross(Vecteur a, Vecteur b) {
    return a.x * b.y - a.y * b.x;
}
double cosine(Vecteur a, Vecteur b) {
    return dot(a, b) / a.abs() / b.abs();
}
double arg(Vecteur a, Vecteur b) {
    return acos(cosine(a, b));
}
```

```
double area(Vecteur a, Vecteur b, Vecteur c) {
    return cross(b - a, c - a) / 2.0;
}
double dist2(Vecteur a, Vecteur b) {
    return std::pow(a.x - b.x, 2) + std::pow(a.y - b.y, 2);
}
double dist(Vecteur a, Vecteur b) {
    return std::sqrt(dist2(a, b));
}
using Point = Vecteur;
```