University of Calgary 2021 Reference Manual v0.1

Steven Hwang, Haohu Shen, Chris Sun

Contents

1 Constants

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef unsigned long long ull;
typedef long double ld;
typedef pair<ll,ll> ii;
typedef vector<ll> vi;
typedef vector<vi> vvi;
typedef complex<double> cd;
const ll MOD = 1e9+7;
const int INF = 1e9;
const ll LLINF = 4e18;
const double EPS = 1e-9;
const double M_EPS = 1.0 + 1e-9;

default_random_engine rng((random_device())());

inline int ceiling(int num, int div){
    return (num + div - 1) / div;
}
```

2 Data Structures

2.1 Disjoint Set Union

Average $O(\alpha(n))$ for all operations

```
namespace DSU {
vector<int> father, Size;
void init(int n) {
    father.resize(n);
    iota(father.begin(), father.end(), 0);
    Size.resize(n, 1);
}
int find(int x) {
    if (father[x] != x) father[x] = find(father[x]);
    return father[x];
}
void merge(int x, int y) {
    x = find(x), y = find(y);
    if (x == y) return;
    if (Size[x] > Size[y]) swap(x, y);
    Size[y] += Size[x];
    father[x] = y;
}
bool isSameSet(int x, int y){return find(x) == find(y);}}
```

2.2 Fenwick Tree

$O(\log n)$ for all operations.

```
struct fenwick_tree
{
    vi ft;
    fenwick_tree(int size){ft.assign(size + 1, 0);}
    ll rsq(int i){
        ll sum = 0; for(; i; i -= (i &(-i))) sum += ft[i];
        return sum;}
    ll rsq(int a, int b){
        return rsq(b) - (a == 1 ? 0 : rsq(a - 1));}
    void adjust(int i, ll amount){
        for(; i < ft.size(); i+=(i & (-i))) ft[i]+=amount;}
};
```

2.3 Segment Tree

An implementation of Segment Tree, it takes $O(n)$ to build, each query/update takes $O(\log n)$

```
struct SegmentTree {
    // MAX: numeric_limits<int>::min()
    // MIN: numeric_limits<int>::max()
    static const int INF = 0x3f3f3f3f;
    vector<int> st, lz;
    int n;
    void build(int p, int l, int r, const vector<int> &A) {
```

```
        if (l == r) { st[p] = A[l]; return; }
        build(2*p, l, (l+r)/2, A);
        build(2*p+1, (l+r)/2+1, r, A);

        // RMQ -> min/max, RSQ -> +
        st[p] = min(st[2*p], st[2*p+1]);
    }
    SegmentTree(vector<int> &A) {
        n = (int)A.size();
        st.resize(n << 2);
        lz.resize(n << 2);

        // 'p' is id of the tree
        // which starts from root = 1
        build(1, 0, n - 1, A);
    }
    void push(int p, int l, int r) {
        if (lz[p]) {
            // RMQ -> update: = lz[p]
            // RMQ -> increment: += lz[p]
            // RSQ -> update: = (r-l+1)*lz[p]
            // RSQ -> increment: += (r-l+1)*lz[p]
            st[p] = lz[p];

            // update: =, increment +=
            if(l!=r) lz[2*p] = lz[2*p+1] = lz[p];
            lz[p] = 0;
        }
    }
    int query(int p, int l, int r, int i, int j) {
        push(p, l, r);
        // RMQ -> INF, RSQ -> 0
        if (r < i or l > j) return INF;
        if (l >= i and r <= j) return st[p];

        // RMQ -> min/max, RSQ -> +
        return min(query(2*p, l, (l+r)/2, i, j),
                   query(2*p+1, (l+r)/2+1, r, i, j));
    }
    void update(int p, int l, int r, int i, int j, int v) {
        push(p, l, r);
        if (r < i or l > j) return;
        if (l >= i and r <= j) { lz[p] = v; push(p, l, r); return; }
        update(2*p, l, (l+r)/2, i, j, v);
        update(2*p+1, (l+r)/2+1, r, i, j, v);

        // RMQ -> min/max, RSQ -> +
        st[p] = min(st[2*p], st[2*p+1]);
    }
};

// Usage:
int main() {
    vector<int> p = {6,1,1,5,6,1,9};
    SegmentTree st(p);
    // Update p[1] to p[3] as 120
    st.update(1, 0, st.n - 1, 1, 3, 120);
    // Query the minimal value from p[0] to p[4]
    cout << st.query(1, 0, st.n - 1, 0, 4) << '\n';
    return 0;
}
```

2.4 Sparse Table

Pre-process takes $O(n \log n)$, each query takes $O(1)$, the array cannot be changed. When the query is asking for range gcd, suppose a single gcd takes $O(\log w)$, then the sparse table takes $O(n(\log n + \log w))$ to pre-process and each query takes $O(\log w)$.

```
struct ST {
    vector<vector<ll>> spt;
    vector<ll> Log2;
    int n, logn;
    explicit ST(int n) : n(n) {
        Log2.resize(n + 5);
        Log2[1] = 0;
        Log2[2] = 1;
        for (int i = 3; i < n + 5; ++i)
            Log2[i] = Log2[i >> 1] + 1;
        logn = floor(log2(n) + 2);
        spt.resize(n + 5, vector<ll>(logn));
    }
    // input an array of n elements
    void input() {
        for (int i = 1; i <= n; ++i) {
            cin >> spt[i][0];
        }
    }
    // Change the operator between
    // spt[i][j-1] and spt[i+(1<<(j-1))][j-1]
    // if you want to have range minimum/gcd/
    // lcm/or/and query
    void build() {
        for (int j = 1; j <= logn; ++j)
            for (int i = 1; i + (1 << j) - 1 <= n; ++i)
                spt[i][j] = max(spt[i][j-1],
                spt[i+(1<<(j-1))][j-1]);
    }
    // Change the operator between
    // spt[l][s] and spt[r-(1<<s)+1][s]
    // if you want to have range minimum/gcd/
    // lcm/or/and query
    ll query(int l, int r) {
        int s = Log2[r-l+1];
        ll ans = max(spt[l][s], spt[r-(1<<s)+1][s]);
        return ans;
    }
};

// Usage:
int main() {
    int n, m, l, r;
    cin >> n >> m;
    ST st(n);
    st.input();
    st.build();
    while (m--) {
        cin >> l >> r;
        if (l > r) swap(l, r);
        cout << st.query(l, r) << '\n';
    }
    return 0;
}
```

2.5 2D Prefix Sum

Pre-process takes $O(nm)$, each query takes $O(1)$

```
namespace PrefixSum2D {
    vector<vector<ll> > pre;
    int n, m;

    void init(const vector<vector<ll> > &A) {

        if (A.empty() || A.front().empty())
            return;
        n = (int) A.size();
        m = (int) A.front().size();
        decltype(pre)().swap(pre);
        pre.resize(n, vector<ll>(m));

        pre[0][0] = A[0][0];
        for (int i = 1; i < m; ++i)
            pre[0][i] = pre[0][i - 1] + A[0][i];
        for (int i = 1; i < n; ++i)
            pre[i][0] = pre[i - 1][0] + A[i][0];
        for (int i = 1; i < n; ++i)
            for (int j = 1; j < m; ++j)
                pre[i][j] = pre[i - 1][j] + pre[i][j - 1]
                - pre[i - 1][j - 1] + A[i][j];
    }

    ll rangeSum(int r0, int c0, int r1, int c1) {
        if (r0 == 0 && c0 == 0)
```

```
            return pre[r1][c1];
        else if (r0 == 0 && c0 != 0)
            return pre[r1][c1] - pre[r1][c0 - 1];
        else if (r0 != 0 && c0 == 0)
            return pre[r1][c1] - pre[r0 - 1][c1];
        return pre[r1][c1] - pre[r0 - 1][c1]
             - pre[r1][c0 - 1] + pre[r0 - 1][c0 - 1];
    }
}
```

3 Graph Theory

3.1 Tree

3.1.1 Least Common Ancestor

Tarjan's Offline Algorithm, preprocess $O(m+n)$, each query cost $O(1)$

```
namespace LCA0 {
    vector<vector<int> > adj;
    vector<int> ancestor;
    vector<bool> vis;
    vector<vector<int> > queries;
    unordered_map<pair<int, int>, int> unmap;
    void Tarjan(int u) {
        vis[u] = true;
        ancestor[u] = u;
        for (const auto &v : adj[u]) {
            if (!vis[v]) {
                Tarjan(v);
                DSU::merge(u, v);
                ancestor[DSU::find(u)] = u;
            }
        }
        for (const auto v : queries[u]) {
            if (vis[v]) {
                unmap[{u, v}] = ancestor[DSU::find(v)];
                unmap[{v, u}] = ancestor[DSU::find(v)];
            }
        }
    }
    void init(int n) {
        DSU::init(n);
        ancestor.resize(n + 5);
        vis.resize(n + 5, false);
        adj.resize(n + 5);
        queries.resize(n + 5);
    }
    // Usage:
    // n nodes, (n - 1) edges, the root is s, m queries,
    // for each query, u and v are given,
    // print lca of u and v
    int main() {
        int n, m, s, u, v;
        cin >> n >> m >> s;
        LCA0::init(n);
        for (int i = 1; i <= n - 1; ++i) {
            cin >> u >> v;
            LCA0::adj[u].emplace_back(v);
            LCA0::adj[v].emplace_back(u);
        }
        vector<ii> query_list;
        for (int i = 1; i <= m; ++i) {
            cin >> u >> v;
            query_list.emplace_back(u, v);
            LCA0::queries[u].emplace_back(v);
            LCA0::queries[v].emplace_back(u);
        }
        LCA0::Tarjan(s);    // Run Tarjan from root
        // output
        for (const auto &[u, v] : query_list) {
            cout << LCA0::unmap[{u, v}] << '\n';
        }
        return 0;
    }
}
```

Obtain LCA by binary lifting, Preprocess $O(n \log n)$, each query costs $O(\log n)$.

```
namespace LCA1 {
    int n, l;
    vector<vector<int>> adj;
    int timer;
    vector<int> tin, tout;
    vector<vector<int>> up;
    void dfs(int v, int p) {
        tin[v] = ++timer;
        up[v][0] = p;
        for (int i = 1; i <= l; ++i)
            up[v][i] = up[up[v][i-1]][i-1];

        for (int u : adj[v]) {
            if (u != p)
                dfs(u, v);
        }
        tout[v] = ++timer;
    }
    bool is_ancestor(int u, int v) {
        return tin[u] <= tin[v] && tout[u] >= tout[v];
    }
    int lca(int u, int v) {
        if (is_ancestor(u, v)) return u;
        if (is_ancestor(v, u)) return v;
        for (int i = l; i >= 0; --i) {
            if (!is_ancestor(up[u][i], v))
                u = up[u][i];
        }
        return up[u][0];
    }
    // Assume node id starts from 0
    void preprocess(int number_of_nodes) {
        n = number_of_nodes;
        tin.resize(n);
        tout.resize(n);
        adj.resize(n);
        timer = 0;
        l = ceil(log2(n));
        up.resize(n, vector<int>(l + 1));
    }
    void init(int root) { dfs(root, root); }
}
```

3.1.2 Prefix Sum on Tree

```
// Prefix sum of edges' weights on a rooted tree
namespace PrefixSumTree0 {
    vector<ll> pre;
    vector<vector<pair<int, ll> > > adj;
    void init(int n) {
        adj.resize(n);
        pre.resize(n);
    }
    void dfs(int u, int father_of_u, ll currSum) {
        for (const auto &[v, w] : adj[u]) {
            if (v != father_of_u) {
                pre[v] = currSum + w;
                dfs(v, u, pre[v]);
            }
        }
    }
    // Query the distance from u to v:
    // ll dist(int u, int v)
    // { return pre[u] + pre[v] - 2*pre[lca(u, v)]; }
    // Usage
    int main() {
        int n, root, m, u, v, q; ll w;
        cin >> n >> root >> m;
        init(n);
        while (m--) {
            cin >> u >> v >> w;
            adj[u].emplace_back(v, w);
```

```
            adj[v].emplace_back(u, w);
        }
        int dummy_node = -1;
        dfs(root, dummy_node, 0);
        while (q--) {
            cin >> u >> v;
            // cout << dist(u, v) << '\n';
        }
        return 0;
    }
}
// Prefix sum of nodes' weights on a rooted tree
namespace PrefixSumTree1 {
    vector<ll> pre;
    vector<vector<int> > adj;
    vector<ll> weight;
    vector<int> father;

    void init(int n) {
        pre.resize(n);
        adj.resize(n);
        weight.resize(n);
        father.resize(n);
    }

    void dfs(int u, int father_of_u, ll currSum) {
        pre[u] = currSum + weight[u];
        for (const auto &v : adj[u]) {
            if (v != father_of_u) {
                dfs(v, u, pre[u]);
                father[v] = u;
            }
        }
    }

    // Query the distance from u to v:
//    ll dist(int u, int v) {
//        int Lca = lca(u, v);
//        // Case 1: lca is root
//        if (father[Lca] == -1) {
//            return pre[u] + pre[v] - pre[Lca];
//        }
//        // Case 2: otherwise
//        return pre[u]+pre[v]-pre[Lca]-pre[father[Lca]];
//    }
    // Usage
    int main() {
        int n, root, m, u, v, q;
        cin >> n >> root >> m;
        init(n);
        for (auto &i : weight) cin >> i;
        while (m--) {
            cin >> u >> v;
            adj[u].emplace_back(v);
            adj[v].emplace_back(u);
        }
        father[root] = -1;
        int dummy_node = -1;
        dfs(root, dummy_node, 0);
        while (q--) {
            cin >> u >> v;
            // cout << dist(u, v) << '\n';
        }
        return 0;
    }
}
```

3.1.3 Tree's Diameter

Obtain the tree diameter by DP in $O(n)$

```
namespace TreeDiameter {
    // the longest distance each node can reach
    vector<int> d1;
    // the second longest distance each node can reach
    vector<int> d2;
    vector<vector<int> > adj;
    void init(int n) {
        adj.resize(n);
        d1.resize(n);
        d2.resize(n);
    }
    void dfs(int u, int father_of_u) {
        d1.at(u) = 0;
        d2.at(u) = 0;
        for (const auto &v : adj.at(u)) {
            if (v == father_of_u) continue;
            dfs(v, u);
            int temp = d1.at(v) + 1;
            if (temp > d1.at(u)) {
                d2.at(u) = d1.at(u);
                d1.at(u) = temp;
            } else if (temp > d2.at(u)) {
                d2.at(u) = temp;
            }
        }
    }
    // Usage:
    int main() {
        int n, m, u, v;
        cin >> n >> m;
        init(n);
        while (m--) {
            cin >> u >> v;
            adj[u].emplace_back(v);
            adj[v].emplace_back(u);
        }
        // randomly pick one node
        // in the graph as root
        dfs(0, -1);
        int diameter = -1;
        for (int i = 0; i < n; ++i) {
            diameter = max(diameter, d1.at(i)+d2.at(i));
        }
        return 0;
    }
}
```

3.1.4 Tree's Center

3.1.5 DSU on Tree

3.1.6 Heavy Light Decomposition

3.2 Spanning Tree

3.2.1 Minimum Spanning Tree

Obtain the MST by Kruskal's algorithm and DSU, takes $O(m \log n)$. To obtain the Maximum Spanning Tree, make all weights negative and run Kruskal.

```
namespace MST_Kruskal {
    struct Edge {
        int u, v;
        int w;
        explicit Edge(int u, int v, int w):u(u),v(v),w(w){}
        Edge() = default;
    };
    vector<Edge> kruskal(vector<Edge> E, int n) {
        // initialize a DSU
        DSU::init(n);
        int index = 0;
        // sort by the edge's weight in increasing order
        sort(E.begin(), E.end(),
             [](const Edge &lhs, const Edge &rhs) {
                 return (lhs.w < rhs.w); });
        vector<Edge> minimum_spanning_tree;
        // to calculate the total weight of the MST
        // int cost = 0;
        for (const auto &[u, v, w] : E) {
            if (!DSU::is_same_group(u, v)) {
```

```
                DSU::merge(u, v);
                minimum_spanning_tree.
                emplace_back(Edge(u, v, w));
                // cost += w;
            }
        }
        return minimum_spanning_tree;
    }
}
```

Obtain the MST by Prim's algorithm, takes $O(m \log n)$

```
namespace MST_Prim {
    vector<vector<pair<int, ll> > > AL;
    vector<bool> taken;
    priority_queue<pair<ll, int> > pq;
    int n;

    void process(int u) {
        taken[u] = true;
        for (auto &[v, w] : AL[u])
            if (!taken[v])
                pq.push({-w, -v});
    }

    ll mst_cost = 0;
    int num_taken = 0;

    void reset() {
        mst_cost = 0;
        num_taken = 0;
        decltype(pq)().swap(pq);
        vector<bool>().swap(taken);
        decltype(AL)().swap(AL);
        n = 0;
    }

    void prim(int source) {
        taken.resize(n, false);
        process(source);
        while (!pq.empty()) {
            auto[w, u] = pq.top();
            pq.pop();
            w = -w;
            u = -u;
            if (taken[u]) continue;
            mst_cost += w;
            process(u);
            ++num_taken;
            if (num_taken == n - 1) break;
        }
    }
}
```

3.2.2 Directed Minimum Spanning Tree

We know that it takes $O(mn)$ to obtain Directed Minimum Spanning Tree using Edmond's algorithm, but how to obtain DMST if the tree is unrooted?

- Suppose the node id in G starts from 1
- Create a virtual node 0 in G.
- Let's say $sum(w)$ is the sum of edges added.
- Place directed edges from node 0 to all other nodes, the weight of each edge is $sum(w) + 1$.
- Run the algorithm, suppose the result is s.
- If $s \geq 2sum(w) + 1$, the DMST does not exists.
- Otherwise, the sum weight of DMST is $s - sum(w) - 1$.

```
namespace DMST {
    constexpr ll INF = 0x3f3f3f3f3f3f3f;
    struct Edge {
        Edge(int u, int v, ll w) : u(u), v(v), w(w){}
        Edge() = default;
        int u, v;
        ll w;
    };
    vector<Edge> edges;
    vector<int> pre;
    vector<int> id;
    vector<int> visit;
    vector<ll> in;

    void reset() {
        vector<Edge>().swap(edges);
        vector<int>().swap(pre);
        vector<int>().swap(id);
        vector<int>().swap(visit);
        vector<ll>().swap(in);
    }

    // return the sum of MDST
    // or -INF if there is no MDST
    ll Edmonds(int root, int number_of_nodes) {
        // init
        int number_of_edges = (int)edges.size();
        pre.resize(number_of_nodes + 5);
        id.resize(number_of_nodes + 5);
        visit.resize(number_of_nodes + 5);
        in.resize(number_of_nodes + 5);

        ll result = 0;
        int u, v, tn;
        while (true) {
            for (int i = 0; i < number_of_nodes; ++i) {
                in.at(i) = INF;
            }
            for (int i = 0; i < number_of_edges; ++i) {
                if (edges.at(i).u != edges.at(i).v &&
                    edges.at(i).w < in.at(edges.at(i).v)) {
                    pre.at(edges.at(i).v) = edges.at(i).u;
                    in.at(edges.at(i).v) = edges.at(i).w;
                }
            }
            for (int i = 0; i < number_of_nodes; ++i) {
                if (i != root && in.at(i) == INF) {
                    return -INF;    // No DMST exists
                }
            }
            tn = 0;
            fill(id.begin(), id.end(), -1);
            fill(visit.begin(), visit.end(), -1);
            in.at(root) = 0;
            for (int i = 0; i < number_of_nodes; ++i) {
                result += in.at(i);
                v = i;
                while (visit.at(v) != i
                    && id.at(v) == -1
                    && v != root) {
                    visit.at(v) = i;
                    v = pre.at(v);
                }
                if (v != root && id.at(v) == -1) {
                    for (u = pre.at(v);
                        u != v;
                        u = pre.at(u)) {
                        id.at(u) = tn;
                    }
                    id.at(v) = tn;
                    ++tn;
                }
            }
            if (tn == 0) {
                break;  // No cycles found
            }
```

```
            for (int i = 0; i < number_of_nodes; ++i) {
                if (id.at(i) == -1) {
                    id.at(i) = tn;
                    ++tn;
                }
            }
            for (int i = 0; i < number_of_edges; ) {
                v = edges.at(i).v;
                edges.at(i).u = id.at(edges.at(i).u);
                edges.at(i).v = id.at(edges.at(i).v);
                if (edges.at(i).u != edges.at(i).v) {
                    edges.at(i).w -= in.at(v);
                    ++i;
                } else {
                    swap(edges.at(i),
                    edges.at(--number_of_edges));
                }
            }
            number_of_nodes = tn;
            root = id.at(root);
        }
        return result;
    }
    // Usage:
    int main() {
        int n, m, r, u, v;
        ll w;
        vector<unordered_map<int, ll> > adj;
        cin >> n >> m >> r;
        --r;
        adj.resize(n + 5);
        for (int i = 0; i < m; ++i) {
            cin >> u >> v >> w;
            if (u == v) continue;
            if (adj.at(u).find(v) == adj.at(u).end()) {
                adj.at(u)[v] = w;
            } else {
                adj.at(u)[v] = min(adj.at(u)[v], w);
            }
        }
        // add edges
        for (int i = 0; i < n; ++i) {
            for (const auto &[j, k] : adj.at(i)) {
                edges.emplace_back(i, j, k);
            }
        }
        ll result = Edmonds(r, n);
        if (result == -INF) {
            cout << -1 << endl;
        } else {
            cout << result << endl;
        }
        return 0;
    }
}
```

3.3 Topological Sort

Topological sort by Kahn's algorithm, takes $O(V + E)$.

```
namespace Toposort {
    vector<vector<int> > AL;
    // assume that the node id starts from 1
    int number_of_nodes;
    vector<int> in_degree;
    // store the result after toposort
    vector<int> result;
    void reset() {
        vector<vector<int> >().swap(AL);
        vector<int>().swap(in_degree);
        number_of_nodes = 0;
        vector<int>().swap(result);
    }
    void init(int n) {
        number_of_nodes = n;
        in_degree.resize(n + 5);
        AL.resize(n + 5);
    }
    // Main logic of Kahn's algorithm: O(V+E)
    // return true if it does not
    // have a cycle, otherwise false
    bool kahn() {
        queue<int> q;
        for (int i = 1; i <= number_of_nodes; ++i) {
            if (in_degree.at(i) == 0) {
                q.push(i);
            }
        }
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            result.emplace_back(u);
            for (const auto &adj_v : AL[u]) {
                --in_degree.at(adj_v);
                if (in_degree.at(adj_v) == 0) {
                    q.push(adj_v);
                }
            }
        }
        return ((int)result.size() == number_of_nodes);
    }

    // Main logic of Kahn's algorithm
    // but the output should be lexicographically
    // smallest among all possible results
    // after toposort: O(VlogV+E)
    bool kahn_with_lexicographically_smallest() {
        std::priority_queue<int, vector<int>, greater<> >q;
        for (int i = 1; i <= number_of_nodes; ++i) {
            if (in_degree.at(i) == 0) {
                q.push(i);
            }
        }
        while (!q.empty()) {
            int u = q.top();
            q.pop();
            result.emplace_back(u);
            for (const auto &adj_v : AL[u]) {
                --in_degree.at(adj_v);
                if (in_degree.at(adj_v) == 0) {
                    q.push(adj_v);
                }
            }
        }
        return ((int)result.size() == number_of_nodes);
    }
}
// Usage:
int main() {
    // Construct the graph
    auto construct_the_graph = [&]() {
        Toposort::init(6);
        Toposort::AL[6].emplace_back(3);
        ++Toposort::in_degree.at(3);
        Toposort::AL[6].emplace_back(1);
        ++Toposort::in_degree.at(1);
        Toposort::AL[5].emplace_back(1);
        ++Toposort::in_degree.at(1);
        Toposort::AL[5].emplace_back(2);
        ++Toposort::in_degree.at(2);
        Toposort::AL[3].emplace_back(4);
        ++Toposort::in_degree.at(4);
        Toposort::AL[4].emplace_back(2);
        ++Toposort::in_degree.at(2);
    };
    // test_for_toposort
    construct_the_graph();
    assert(Toposort::kahn());
    Toposort::reset();
    construct_the_graph();
    assert(Toposort::kahn_with_lexicographically_smallest());
    vector<int> result_ = {5, 6, 1, 3, 4, 2};
    assert(Toposort::result == result_);
    return 0;
}
```

3.4 All Pairs Shortest Path

Floyd Warshall's Algorithm, will find all pairs of shortest path in a graph in $O(n^3)$.

```
namespace APSP0{
    constexpr int INF = 0x3f3f3f3f;
    // adjacency matrix
    vector<vector<int> > am;
    // prev[i][j] is the previous vertex of j
    vector<vector<int> > prev;
    int n;
    void init(int number_of_nodes, bool store_path=false) {
        // reset
        vector<vector<int> >().swap(am);
        n = number_of_nodes;
        am.resize(n, vector<int>(n, INF));
        for (int i = 0; i < n; ++i) {
            // true on most cases
            am[i][i] = 0;
        }
        if (store_path) {
            prev.resize(n, vector<int>(n));
            for (int i = 0; i < n; ++i) {
                // true on most cases
                prev[i][i] = i;
            }
        }
    }
    void floyd_warshall(bool store_path=false){
        // order: k, i, j
        for (int k = 0; k < n; ++k)
            for (int i = 0; i < n; ++i)
                for (int j = 0; j < n; ++j)
                    if (am[i][k]<INF&&am[k][j]<INF) {
                        if (am[i][j]>am[i][k]+am[k][j]) {
                            am[i][j]=am[i][k]+am[k][j];
                            if (store_path) {
                                prev[i][j]=prev[k][j];
                            }
                        }
                    }
        // No need for graph with edges
        // of non-negative weights
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                for (int t = 0; t < n; ++t)
                    if (am[i][t]<INF
                        &&am[t][t]<0
                        &&am[t][j]<INF)
                        am[i][j]=-INF;
    }
    // Query the shortest distance from u to v
    // INF: cant reach -INF: in a negative cycle
    int dist(const int &u, const int &v) {
        return am[u][v];
    }
    // Print the path from u to v
    void print_path(const int &u, const int &v) {
        if (dist(u, v) == INF) {
            cout << "\nthe final path does not exist\n";
            return;
        }
        if (u != v) {
            print_path(u, prev[u][v]);
        }
        cout << v << ' ';
    }
    // Usage
    int main() {
        int number_of_nodes, m, u, v, w, s, d;
        cin >> number_of_nodes >> m;
        init(number_of_nodes, true);
        while (m--) {
            cin >> u >> v >> w;
            am[u][v] = w;
            prev[u][v] = u;
        }
        cin >> s >> d;
        floyd_warshall(true);
        print_path(s, d);
        return 0;
    }
}
```

3.5 Single Source Shortest Path

3.5.1 Dijkstra

Implementation of Dijkstra using adjacency list and priority queue optimized, runtime takes $O(m \log m)$

```
namespace SSSP_Dijkstra {
    constexpr int INF = 0x3f3f3f3f;
    // The adjacency list of the graph
    vector<vector<pair<int, int> > > adj;
    int number_of_nodes;
    vector<int> prev;
    void
    init(int n, bool store_path = false) {
        number_of_nodes = n;
        // initialize the containers
        adj.resize(number_of_nodes);
        if (store_path) {
            prev.resize(number_of_nodes, -1);
        }
    }
    void
    reset(bool store_path = false) {
        if (store_path) {
            fill(prev.begin(), prev.end(), -1);
        }
    }
    void
    add_edge(int u, int v, int w) {
        adj.at(u).emplace_back(make_pair(v, w));
    }
    vector<int>
    dijkstra(int source, bool store_path = false) {
        // dist.at(u) = the shortest distance
        // from source to u
        vector<int> dist(number_of_nodes, INF);
        dist.at(source) = 0;
        std::priority_queue<
        pair<int, int>,
        vector<pair<int, int> >,
        greater<> > pq;
        pq.push(make_pair(dist.at(source), source));
        while (!pq.empty()) {
            int d_v = pq.top().first;
            int v = pq.top().second;
            pq.pop();
            if (d_v != dist.at(v)) {
                continue;
            }
            for (const auto &[to, weight] : adj.at(v)) {
                if (dist.at(to) > dist.at(v) + weight) {
                    dist.at(to) = dist.at(v) + weight;
                    if (store_path) {
                        prev.at(to) = v;
                    }
                    pq.push({dist.at(to), to});
                }
            }
        }
        return dist;
    }
    vector<int>
    get_path(int destination) {
        vector<int> path;
        for (; destination != -1;
            destination = prev.at(destination)) {
            path.emplace_back(destination);
        }
        reverse(path.begin(), path.end());
```

```
        return path;
    }
}
```

3.5.2 SPFA

Shortest Path Faster Algorithm is an implementation of Bellman-Ford's algorithm with queue optimized, the runtime is $O(mn)$. The algorithm is also called Bellman-Ford-Moore's algorithm.

```
// If you want to query the longest path
// from a single source, change INF to -INF and
// change the condition of relaxing from < to >
namespace SSSP_SPFA {
    constexpr int INF = 0x3f3f3f3f;
    // The adjacency list of the graph
    vector<vector<pair<int, int> > > adj;
    int source;
    int number_of_nodes;
    vector<int> dis;
    void
    reset() {
        decltype(adj)().swap(adj);
        vector<int>().swap(dis);
        number_of_nodes = 0;
        source = 0;
    }
    void
    init(int n, int s) {
        source = s;
        number_of_nodes = n;
        adj.resize(number_of_nodes + 5);
    }
    vector<int> pre;
    bool
    spfa(bool store_path = false) {
        if (store_path) {
            pre.resize(number_of_nodes + 5, -1);
        }
        dis.resize(number_of_nodes + 5, INF);
        dis.at(source) = 0;
        vector<int> cnt(number_of_nodes + 5);
        vector<bool> in_queue(number_of_nodes + 5, false);
        queue<int> q;
        q.push(source);
        in_queue.at(source) = true;
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            in_queue.at(u) = false;
            for (const auto &[v, w] : adj.at(u)) {
                if (dis.at(u) + w < dis.at(v)) {
                    dis.at(v) = dis.at(u) + w;
                    if (store_path) {
                        pre.at(v) = u;
                    }
                    if (!in_queue.at(v)) {
                        q.push(v);
                        in_queue.at(v) = true;
                        ++cnt.at(v);
                        if (cnt.at(v) > number_of_nodes){
                            // a negative cycle exists
                            return false;
                        }
                    }
                }
            }
        }
        return true;
    }
    vector<int>
    get_path(int dest) {
        vector<int> path;
        for (; dest != -1; dest = pre.at(dest)) {
            path.emplace_back(dest);
        }
        reverse(path.begin(), path.end());
        return path;
    }
}
```

3.6 Connectivity

3.6.1 Strong Connected Components

Obtain all SCCs in a directed graph in $O(m+n)$ by Tarjan's algorithm

```
// Assume that the node id starts from 1 and the
// index of a strongly connected component (SCC)
// also starts from 1
namespace SCC_Tarjan {

    // adjacency list of the graph
    vector<vector<int> > G;
    int number_of_nodes;
    int number_of_scc;
    int current_timestamp;
    stack<int> s;
    vector<bool> vis;
    vector<int> dfs_rank;
    vector<int> low_link;

    // scc.at(id) is the index of the strongly connected
    // component that the node id belongs to
    vector<int> scc;

    // size_of_scc.at(id) is the size of the strongly
    // connected component whose index is id
    vector<int> size_of_scc;

    // reset all containers
    void reset() {
        decltype(G)().swap(G);
        stack<int>().swap(s);
        vector<bool>().swap(vis);
        vector<int>().swap(dfs_rank);
        vector<int>().swap(low_link);
        vector<int>().swap(scc);
        vector<int>().swap(size_of_scc);
    }

    // Initialize all global variables in the namespace
    void init(int n) {
        number_of_nodes = n;
        number_of_scc = 0;
        // NOTICE: starts from 1 if the node id starts from 1
        current_timestamp = 1;
        G.resize(number_of_nodes + 5);
        vis.resize(number_of_nodes + 5, false);
        dfs_rank.resize(number_of_nodes + 5, 0);
        low_link.resize(number_of_nodes + 5, 0);
        scc.resize(number_of_nodes + 5, 0);
        size_of_scc.resize(number_of_nodes + 5, 0);
    }

    void Tarjan(int u) {
        dfs_rank.at(u) = current_timestamp;
        low_link.at(u) = current_timestamp;
        ++current_timestamp;
        s.push(u);
        vis.at(u) = true;
        for (const auto &v : G[u]) {
            if (!dfs_rank[v]) {
                Tarjan(v);
                low_link.at(u) =
                min(low_link.at(u), low_link.at(v));
            } else if (vis.at(v)) {
                low_link.at(u) =
                min(low_link.at(u), dfs_rank.at(v));
            }
        }
        if (low_link.at(u) == dfs_rank.at(u)) {
```

```
            ++number_of_scc;
            while (s.top() != u) {
                int top_id = s.top();
                // Paint top_id
                s.pop();
                scc.at(top_id) = number_of_scc;
                ++size_of_scc.at(number_of_scc);
                vis.at(top_id) = false;
            }
            // Paint u
            s.pop();
            scc.at(u) = number_of_scc;
            ++size_of_scc.at(number_of_scc);
            vis.at(u) = false;
        }
    }

    // Usage:
    int main() {
        int n, m, u, v;
        cin >> n >> m;
        init(n);
        while (m--) {
            cin >> u >> v;
            G[u].emplace_back(v);
        }
        // Run Tarjan's SCC algorithm
        for (int i = 1; i <= n; ++i) {
            // Process node i if the node i
            // has not been visited
            if (!dfs_rank.at(i)) {
                Tarjan(i);
            }
        }
        return 0;
    }
}
```

3.6.2 Cut Vertices And Bridges

Obtain all cut vertices and bridges in an undirected graph in $O(m+n)$

```
namespace CutVertexAndBridges {

    int n; // number of nodes
    vector<vector<int>> adj; // adjacency list of graph
    vector<bool> visited;
    vector<bool> isCutVertex;
    vector<pair<int, int> > bridges;
    vector<int> tin, low;
    int timer;

    void init(int number_of_nodes) {
        n = number_of_nodes;
        adj.resize(n + 5);
        visited.resize(n + 5, false);
        isCutVertex.resize(n + 5, false);
        tin.resize(n + 5);
        low.resize(n + 5);
    }

    void dfs(int u, int p = -1) {
        visited[u] = true;
        tin[u] = low[u] = timer++;
        int children = 0;
        for (const auto v : adj[u]) {
            if (v == p) continue;
            if (visited[v]) {
                low[u] = min(low[u], tin[v]);
            } else {
                dfs(v, u);
                low[u] = min(low[u], low[v]);
                if (low[v] > tin[u]) {
                    if (u > v) {
                        bridges.emplace_back(v, u);
                    } else {
                        bridges.emplace_back(u, v);
                    }
                }
                if (low[v] >= tin[u] && p != -1)
                    isCutVertex[u] = true;
                ++children;
            }
        }
        if (p == -1 && children > 1)
            isCutVertex[u] = true;
    }

    void findCutVerticesAndBridges() {
        timer = 0;
        visited.resize(n, false);
        tin.resize(n, -1);
        low.resize(n, -1);
        isCutVertex.resize(n, false);
        vector<pair<int, int> >().swap(bridges);
        // Assume node id starts from 0
        for (int i = 0; i < n; ++i) {
            if (!visited[i]) dfs(i);
        }
    }
}
```

3.6.3 Edges Bi-connected Components (eBCC)

- Identify all bridges in an undirected graph G
- Remove all bridges in G, we have G'
- Rescan G's adjacency list, ignore all bridges, use DSU to get CCs in G'
- Each CC in G' is an eBCC

3.6.4 Vertices Bi-connected Components (vBCC)

3.7 Cycles

3.7.1 Minimum Cycle

```
namespace MinimumWeightCycleUndirectedGraph {
    constexpr int INF = 0x3f3f3f3f;
    vector<vector<int> > adj_matrix;
    int number_of_nodes;
    // node id starts from 1
    void init(int n) {
        number_of_nodes = n;
        adj_matrix.resize(number_of_nodes + 5,
                          vector<int>
                          (number_of_nodes+5,INF));
        for (int i = 1; i <= number_of_nodes; ++i) {
            adj_matrix[i][i] = 0;
        }
    }
    void add_edge(int u, int v, int w) {
        adj_matrix[u][v] = w;
        adj_matrix[v][u] = w;
    }
    // Return false if no such cycle exists
    bool floyd(ll &answer) {
        vector<vector<int> > dis(adj_matrix);
        answer = INF;
        for (int k = 1; k <= number_of_nodes; ++k) {
            for (int i = 1; i < k; ++i) {
                for (int j = 1; j < i; ++j) {
                    answer = min(answer, dis[i][j] * 1LL +
                                         adj_matrix[i][k] +
                                         adj_matrix[k][j]);
                }
            }
```

```
            for (int i = 1; i <= number_of_nodes; ++i) {
                for (int j = 1; j <= number_of_nodes; ++j) {
                    dis[i][j]=
                    min(dis[i][j],dis[i][k]+dis[k][j]);
                }
            }
        }
        return answer < INF;
    }
}

// Find minimum weight cycle in a directed graph
// using Dijkstra with flag
namespace MinimumWeightCycleDirectedGraph {

    constexpr int INF = 0x3f3f3f3f;

    // max nodes in the graph
    constexpr int MAXN = 505;

    // The adjacency list of the graph
    vector<vector<pair<int, int> > > adj;
    int number_of_nodes;
    bitset<MAXN> vis;

    void
    init(int n) {
        number_of_nodes = n;
        // initialize the containers
        adj.resize(number_of_nodes + 5);
    }

    void
    add_edge(int u, int v, int w) {
        adj.at(u).emplace_back(v, w);
    }

    vector<int>
    dijkstra(int source) {
        vis.reset();
        bool flag = true;
        vector<int> dis(number_of_nodes + 5, INF);
        dis.at(source) = 0;
        std::priority_queue<
        pair<int, int>,
        vector<pair<int, int> >,
        greater<> > pq;
        pq.push(make_pair(dis.at(source), source));
        while (!pq.empty()) {
            int v = pq.top().second;
            pq.pop();
            if (vis[v]) {
                continue;
            }
            vis[v] = true;
            for (const auto &[to, w] : adj.at(v)) {
                if (dis.at(to) > dis.at(v) + w) {
                    dis.at(to) = dis.at(v) + w;
                    if (!vis[to]) {
                        pq.push(make_pair(dis.at(to), to));
                    }
                }
            }
            // reset the start point
            if (flag) {
                vis[source] = false;
                dis.at(source) = INF;
                flag = false;
            }
        }
        return dis;
    }

    // Usage:
    int main() {
        // suppose in this case we have 400 nodes
        int n = 400;
        init(n);
        // ... After the construction of the graph,
        // node id starts from 1
        int minimalCycle = INF;
        for (int i = 1; i <= n; ++i) {
            auto dis = dijkstra(i);
            minimalCycle = dis.at(i);
        }
        if (minimalCycle == INF) {
            cout << "No cycle found!" << '\n';
        } else {
            cout << minimalCycle << '\n';
        }
        return 0;
    }
}
```

3.7.2 Transitive Closure

An implementation of Floyd Warshall Algorithm to obtain the transitive closure of a directed graph in $O(N^3/w)$ with bitset optimized

```
template<size_t N>
struct TransitiveClosure {
    bitset<N> reach[N]; // reach[i][i] = true
    // Assume that node id starts from 0
    void floydWarshall(int n) {
        for (int k = 0; k < n; ++k) {
            for (int i = 0; i < n; ++i) {
                if (reach[i][k]) {
                    reach[i] |= reach[k];
                }
            }
        }
    }
    void reset() {
        for (size_t i = 0; i < N; ++i) {
            reach[i].reset();
        }
    }
    bool canReach(int u, int v) {
        return reach[u][v];
    }
};
```

3.8 2-SAT

3.9 Eulerian Cycle/Path

- Eulerian Path: a trail in a finite graph that visits every EDGE exactly once
- Eulerian Circuit: an Eulerian trail that starts and ends on the same vertex
- For an undirected graph G:
  - G has an Eulerian Circuit iff G is connected, G has no vertices with odd degree
  - G has an Eulerian Path iff G is connected, G has 0 or 2 vertices with odd degree
- For a directed graph G:
  - G has an Eulerian Circuit iff G is a single SCC and for all vertex v, in_degree[v] = out_degree[v]
  - G has an Eulerian Path iff Suppose its underlying undirected graph is G', for G':
    - G' is a single CC
    - there is at most 1 vertex v such that in_degree[v] - out_degree[v] = 1

    * there is at most 1 vertex v such that out_degree[v] - in_degree[v] = 1
    * for other vertices v, in_degree[v] = out_degree[v]

- We use Hierholzer's algorithm to find the Eulerian Circuit/Path:
  - If we confirms G contains an Eulerian Path, then:
    * if G is undirected, then 'startVertex' is one of the vertex with odd degree
    * if G is directed, then out_degree[startVertex] - in_degree[startVertex] = 1
  - If we confirms G contains an Eulerian Circuit, then 'startVertex' can be any of vertices.
- Sort the adjacency list if you need to find the lexicographically smallest/largest Eulerian Path/Circuit

```
// Hierholzer for directed graphs
vector<int> Hierholzer(int startVertex,
vector<deque<int>> &adjList) {
    stack<int> path;
    vector<int> circuit;
    int current = startVertex;
    path.push(startVertex);
    while (!path.empty()) {
        if (!adjList.at(current).empty()) {
            path.push(current);
            int next = adjList.at(current).front();
            adjList.at(current).pop_front();
            current = next;
        } else {
            circuit.emplace_back(current);
            current = path.top();
            path.pop();
        }
    }
    reverse(circuit.begin(), circuit.end());
    return circuit;
}

// Hierholzer for undirected graphs
// Once an edge (u,v) is added to the graph
// We update 'stats' by ++stats[u][v], ++stats[v][u]
unordered_map<int, unordered_map<int, int> > stats;
vector<int> Hierholzer2(int startVertex,
vector<deque<int>> &adjList) {
    stack<int> path;
    vector<int> circuit;
    int current = startVertex;
    path.push(startVertex);
    while (!path.empty()) {
        if (!adjList.at(current).empty()) {
            path.push(current);
            int next = adjList.at(current).front();
            adjList.at(current).pop_front();
            // avoid traversing the same edge twice
            if (stats[current][next] > 0) {
                --stats[current][next];
                --stats[next][current];
                current = next;
            }
        } else {
            circuit.emplace_back(current);
            current = path.top();
            path.pop();
        }
    }
    reverse(circuit.begin(), circuit.end());
    return circuit;
}
```

3.10 Hamiltonian Cycle/Path

3.11 Bipartite Graph

3.11.1 Bipartite Check

Check whether an undirected graph is bipartite in $O(m+n)$

```
// Theory: a graph is bipartite
// if and only if it is two-colorable (0/1)
namespace BipartiteCheck {
    constexpr int INF = 0x3f3f3f3f;
    // assume node id starts from 0
    vector<vector<int> > adj;
    vector<int> color;
    int n;
    void init(int number_of_nodes) {
        n = number_of_nodes;
        vector<int>().swap(color);
        color.resize(n + 5, INF);
    }
    // bfs from every node id, which
    // will cover different CCs
    bool bfs() {
        queue<int> q;
        for (int s = 0; s < n; ++s) {
            if (color.at(s) == INF) {
                color.at(s) = 0;
                q.push(s);
            }
            while (!q.empty()) {
                int u = q.front();
                q.pop();
                for (const auto &v : adj.at(u)) {
                    if (color.at(v) == INF) {
                        color.at(v) = 1 - color.at(u);
                        q.push(v);
                    } else if (color.at(v) == color.at(u)) {
                        // Coloring conflict found, exit
                        return false;
                    }
                }
            }
        }
        return true;
    }
    // Another version that bfs from 's', which means
    // we only consider if the CC that contains 's'
    // is bipartite
    bool bfs(int s) {
        bool isBipartite = true;
        queue<int> q;
        color.at(s) = 0;
        q.push(s);
        while (!q.empty() && isBipartite) {
            int u = q.front();
            q.pop();
            for (const auto &v : adj.at(u)) {
                if (color.at(v) == INF) {
                    color.at(v) = 1 - color.at(u);
                    q.push(v);
                } else if (color.at(v) == color.at(u)) {
                    // Coloring conflict found
                    isBipartite = false;
                    break;
                }
            }
        }
        return isBipartite;
    }
}
```

3.12 Stable Marriage Problem

Assume n men, n women. Men are proposers. Gale Shapley's algorithm cost $O(n^2)$

```
namespace SMP {
```

```
    vector<queue<int> > m_pref;
    vector<vector<int> > w_pref;
    // engaged[i]: woman, i: man
    vector<int> engaged;
    queue<int> free_man;
    vector<bool> is_woman_free;
    void reset() {
        vector<queue<int> >().swap(m_pref);
        vector<vector<int> >().swap(w_pref);
        vector<int>().swap(engaged);
        queue<int>().swap(free_man);
        vector<bool>().swap(is_woman_free);
    }
    void init(int n) {
        // man/woman's id start from 1
        m_pref.resize(n + 1);
        w_pref.resize(n + 1, vector<int>(n + 1));
        engaged.resize(n + 1);
        for (int i = 1; i <= n; ++i) free_man.push(i);
        is_woman_free.resize(n + 1, true);
    }
    void galeShapley() {
        while (!free_man.empty()) {
            int m = free_man.front();
            if (m_pref.at(m).empty()) {
                free_man.pop();
                continue;
            }
            // first woman on m's list to whom
            // m has not yet proposed
            int w = m_pref.at(
                    m).front();
            if (is_woman_free.at(w)) {
                engaged.at(w) = m;
                m_pref.at(m).pop();
                free_man.pop();
                is_woman_free.at(w) = false;
            } else {
                int m_ = engaged.at(w);
                if (w_pref.at(w).at(m) >
                    w_pref.at(w).at(m_)) {
                    // w prefers m to m_
                    free_man.push(m_);
                    engaged.at(w) = m;
                    free_man.pop();
                }
                m_pref.at(m).pop();
            }
        }
    }
    // Usage
    int main() {
        int n, val;
        cin >> n;
        init(n);
        // read men's preferences
        for (int i = 1; i <= n; ++i) {
            for (int j = 1; j <= n; ++j) {
                cin >> val;
                m_pref.at(i).push(val);
            }
        }
        // read women's preferences
        for (int i = 1; i <= n; ++i) {
            for (int j = 1; j <= n; ++j) {
                cin >> val;
                w_pref.at(i).at(val) = n + 1 - j;
            }
        }
        galeShapley();
        // print result: m to w
        for (int i = 1; i <= n; ++i) {
            cout << engaged.at(i) << ' ' << i << '\n';
        }
        return 0;
    }
}
```

3.13 Network Flow

3.13.1 Maxflow

3.13.2 Mincut

3.13.3 Mincost Maxflow

Min Cost Max Flow, takes $O(n^2m^2)$

```
// Min Cost Max Flow
namespace MCMF {
    using edge = tuple<int,ll,ll,ll>;
    constexpr ll INF = 1e18;
    int n;
    ll totalCost;
    vector<edge> EL;
    vector<vector<int> > AL;
    vector<ll> d;
    vector<int> last;
    vector<bool> vis;
    void reset() {
        n = 0;
        totalCost = 0;
        decltype(EL)().swap(EL);
        decltype(AL)().swap(AL);
        decltype(d)().swap(d);
        decltype(last)().swap(last);
        decltype(vis)().swap(vis);
    }
    void init(int n_) {
        reset();
        n = n_;
        d.resize(n);
        vis.resize(n);
        AL.resize(n);
        last.resize(n);
    }
    // SPFA to find if there is an augmenting path
    // in residual graph
    bool spfa(int s, int t) {
        fill(d.begin(), d.end(), INF);
        d[s] = 0;
        vis[s] = true;
        queue<int> q;
        q.push(s);
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            vis[u] = false;
            for (const auto &idx : AL[u]) {
                auto &[v, cap, flow, cost] = EL[idx];
                if (cap > flow && d[v] > d[u] + cost) {
                    d[v] = d[u] + cost;
                    if (!vis[v]) {
                        q.push(v);
                        vis[v] = true;
                    }
                }
            }
        }
        return d[t] != INF;
    }
    ll dfs(int u, int t, ll f = INF) {
        if (u == t || f == 0) {
            return f;
        }
        vis[u] = true;
        for (int &i = last[u]; i < (int)AL[u].size(); ++i) {
            auto &[v, cap, flow, cost] = EL[AL[u][i]];
            if (!vis[v] && d[v] == d[u] + cost) {
                if (ll pushed = dfs(v, t, min(f, cap - flow)))
                {
                    totalCost += pushed * cost;
                    flow += pushed;
                    auto &[rv, rcap, rflow, rcost]
                    = EL[AL[u][i]^1];    // back edge
                    rflow -= pushed;
```

```
                    vis[u] = false;
                    return pushed;
                }
            }
        }
        vis[u] = false;
        return 0;
    }
    void addEdge(int u, int v, ll w, ll c,
    bool directed = true) {
        if (u == v) {
            return;
        }
        EL.emplace_back(v, w, 0, c);
        AL[u].emplace_back((int)EL.size() - 1);
        EL.emplace_back(u, directed ? 0 : w, 0, -c);
        AL[v].emplace_back((int)EL.size() - 1);
    }
    pair<ll,ll> mcmf(int s, int t) {
        ll mf = 0;
        while (spfa(s, t)) {
            fill(last.begin(), last.end(), 0);
            while (ll f = dfs(s,t)) {
                mf += f;
            }
        }
        return {mf, totalCost};
    }
}
// Usage: for a directed graph
int main() {
    int n, m, s, t;
    cin >> n >> m >> s >> t;
    MCMF::init(n);
    int u, v;
    ll capacity, cost;
    while (m--) {
        cin >> u >> v >> capacity >> cost;
        MCMF::addEdge(u,v,capacity,cost);
    }
    auto ans = MCMF::mcmf(s,t);
    cout << ans.first << '\n';
    cout << ans.second << '\n';
    return 0;
}
```

3.14 Prufer Code

4 Mathematics

4.1 Binary Greatest Common Divisor

$O(\log a \log b)$

```
ll bingcd(ll a, ll b) {
    if (a < 0 || b < 0) return bingcd(abs(a), abs(b));
    if (!a || !b) return a | b;
    unsigned shift = __builtin_ctz(a | b);
    a >>= __builtin_ctz(a);
    do {
        b >>= __builtin_ctz(b);
        if (a > b) swap(a, b);
        b -= a;
    } while (b); return a << shift; }
```

4.2 Binary Exponentiation

$O(\log b)$ multiplications operations of a

```
ll binpow(ll a, ll b) {
    ll res = 1;
    while (b > 0) {
        if (b & 1) res = res * a;
        a = a * a;
        b >>= 1;}
    return res;}
```

4.3 Modular Multiplication

$O(1)$ runtime to calculate $a \cdot b \bmod m$

```
ll multmod(ll a, ll b, ll m) {
    a = (a % m + m) % m;
    b = (b % m + m) % m;
    return ((a*b -(ll)((ld)a/m*b)*m)%m+m)%m; }
```

4.4 Modular Exponentiation

$O(\log b)$ runtime to calculate $a^b \bmod m$

```
ll powmod(ll a, ll b, ll m) {
    if (m == 1) return 0; ll r;
    for (r = 1, a %= m; b; a = multmod(a,a,m), b >>= 1)
        if (b % 2) r = multmod(r,a,m);
    return r;
}
```

4.5 Extended Euclidean Algorithm

$O(\log(\min(a, b)))$, same as normal gcd.

```
int extEuclid(int a, int b, int &x, int &y) {
  int xx = y = 0;
  int yy = x = 1;
  while (b) {
    int q = a/b;
    tie(a, b) = tuple(b, a%b);
    tie(x, xx) = tuple(xx, x-q*xx);
    tie(y, yy) = tuple(yy, y-q*yy);
  }
  return a;
}
```

4.6 Linear Diophantine Equations

An integral solution for $ax+by = c$ exists if and only if $\gcd(a, b)|c$. Compute x, y such that for a given a, b, $xa + yb = \gcd(a, b)$ in $O(\log(\min(a, b)))$.

4.7 Modular Multiplicative Inverse

Compute $b^{-1} \bmod m$ if $\gcd(b, m) = 1$ in $O(\log(\min(b, m)))$.

```
int modInverse(int b, int m) {
  int x, y;
  int d = extEuclid(b, m, x, y);
  if (d != 1) return -1;
  return ((x%m) + m)%m;
}
```

4.8 Sieves

$O(N \log \log N)$ in upperbound.

```
ll _sieve_size;
bitset<10000010> bs;
vll p;

void sieve(ll upperbound) {
    _sieve_size = upperbound + 1;
    bs.set();
    bs[0] = bs[1] = 0;
    for (ll i = 2; i < _sieve_size; ++i) if (bs[i]) {
        for (ll j = i * i; j < _sieve_size; j += i)
            bs[j] = 0;
        p.push_back(i);
    }
}
```

4.9 Functions Involving Prime Factors

4.9.1 Number of prime factors

$O(\sqrt{N}/\ln\sqrt{N}$ in time, but needs a list of primes p.

```cpp
int numPF(ll N) {
    int ans = 0;
    for (int i = 0; (i < (int)p.size())
        && (p[i]*p[i] <= N); i++)
        while (N%p[i] == 0) { N/= p[i]; ++ans; }
    return ans + (N != 1);
}
```

4.9.2 Number of divisors

$O(\sqrt{N}/\ln\sqrt{N}$ in time but needs a list of primes p.

```cpp
int numDiv(ll N) {
    int ans = 1;
    for (int i = 0; (i < (int)p.size())
        && (p[i]*p[i] <= N); ++i) {
        int power = 0;
        while (N%p[i] == 0) { N /= p[i]; ++power; }
        ans *= power+1;
    }
    return (N != 1) ? 2*ans : ans;
}
```

4.9.3 Sum of Divisors of N

Sum of the divisors of integer $N = a^i \times \cdots \times c^k$ is $\frac{a^{i+1}-1}{a-1} \times \cdots \times \frac{c^{k+1}-1}{c-1}$.
$O(\sqrt{N}/\ln\sqrt{N})$, requires a list p of prime numbers $< \sqrt{N}$

```cpp
ll sumDiv(ll N) {
    ll ans = 1;
    for (int i = 0; (i < (int)p.size())
        && (p[i]*p[i] <= N); ++i) {
        ll multiplier = p[i], total = 1;
        while (N%p[i] == 0) {
            N /= p[i];
            total += multiplier;
            multiplier *= p[i];
        }
        ans *= total;
    }
    if (N != 1) ans *= (N+1);
    return ans;
}
```

4.9.4 Euler's Phi Function

Counts the number of positive integers $< N$ that are relatively prime to N. $\phi(N) = N \times \Pi_{p_i}(1 - \frac{1}{p_i})$.
$O(\sqrt{N}/\ln\sqrt{N})$, requires a list p of prime numbers $< \sqrt{N}$

```cpp
ll EulerPhi(ll N) {
    ll ans = N;
    for (int i = 0; (i < (int)p.size())
        && (p[i]*p[i] <= N); ++i) {
        if (N%p[i] == 0) ans -= ans/p[i];
        while (N%p[i] == 0) N /= p[i];
    }
    if (N != 1) ans -= ans/N;   //last factor
    return ans;
}
```

4.10 Combinatorics

4.10.1 Fibonacci Numbers

Calculate in $O(\log n)$ with formula $fib(n) = (\phi^n - (-\phi)^{-n})/\sqrt{5}$, where $\phi = (\sqrt{5}+1)/2$. Accurate up to $n = 75$ when using double precision. Can also calculate with matrix powers in $O(\log(n))$:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} fib(n+1) & fib(n) \\ fib(n) & fib(n-1) \end{pmatrix}$$

.

4.10.2 Binomial Coefficients

Get a pre-processed array `fact` of $n!\%p$ and calculate $C(n,k)\%p$ in $O(\log(p))$.

```cpp
const in MAX_N = 100010;
const int p = 1e9+7;
ll inv(ll a) {
    return powmod(a, p-2, p);   //modular exponentiation
}
ll fact[MAX_N];
ll C(int n, int k) {
    if (n < k) return 0;
    return (((fact[n] * inv(fact[k])) % p) *
        inv(fact[n-k])) %p;
}
```

4.11 Catalan Numbers

$Cat(n) = \frac{(2n)!}{n!\times n!\times(n+1)}$. Calculate next Catalan number mod p in $O(\log(p))$

```cpp
//inv from binomial coefficents
Cat[n+1] = ((4*n+2)%p * Cat[n]%p * inv(n+2))%p;
```

4.12 Cycle Finding

$O(\mu + \lambda)$ in time, $O(1)$ in space.

```cpp
ii floydCycleFinding(int x0) {
    int t = f(x0), h = f(f(x0));
    while (t !=h) { t = f(t); h = f(f(h)); }
    int mu = 0; h = x0;
    while (t != h) { t = f(t); h = f(h); ++mu; }
    int lambda = 1; h = f(t);
    while (t != h) { h = f(h); ++lambda; }
    return {mu, lambda};
}
```

4.13 Modular Matrix Power

For an $n \times n$ matrix M, takes $O(n^3 \log(p))$ to get M^p.

```cpp
ll MOD = (ll) 1e9 + 7;
const int MAX_N = 2;
struct Matrix { ll mat[MAX_N][MAX_N]; };
ll mod(ll a, ll m) { return ((a%m)+m)%m;}
Matrix matMul(Matrix a, Matrix b) {
    Matrix ans;
    for (int i = 0; i < MAX_N; i++)
        for (int j = 0; j < MAX_N; j++)
            ans.mat[i][j] = 0;
    for (int i = 0; i < MAX_N; i++)
        for (int k = 0; k < MAX_N; k++) {
            if (a.mat[i][k] == 0) continue;
            for (int j = 0; j < MAX_N; ++j) {
                ans.mat[i][j] += mod(a.mat[i][k], MOD)
                    * mod(b.mat[k][j], MOD);
                ans.mat[i][j] = mod(ans.mat[i][j], MOD);
            }
        }
    return ans;
}
Matrix matPow(Matrix base, int p) {
    Matrix ans;
    for (int i = 0; i < MAX_N; i++)
        for (int j = 0; j < MAX_N; j++)
            ans.mat[i][j] = (i == j);
    while (p) {
        if (p&1) ans = matMul(ans, base);
```

```
        base = matMul(base, base);
        p >>= 1;
    }
    return ans;
}
```

4.14 Primality Test

Expected runtime: $O(R(\log(n))^3)$

```
bool miller_rabin_subroutine(ll a, ll n, ll x, ll t) {
    ll result = powmod(a, x, n);
    ll last = result;
    for (int i = 1; i <= t; i++) {
        result = multmod(result, result, n);
        if (result == 1 && last != 1 && last != n - 1)
            return true;
        last = result;
    }
    return result != 1;
}

// Goal: Check if the given number is a prime
// return False if n is not a prime
// return True if n may be a prime
// The rate of inaccuracy is 4^(-R)
// add R if you get an incorrect answer
// also remember adding R will increase the runtime
bool miller_rabin(ll n) {
    constexpr int R = 5;   // rounds
    if (n < 2) return false;
    if (n == 2) return true;
    if ((n & 1) == 0) return false;
    ll x = n - 1, t = 0; ll a;
    while ((x & 1) == 0) { x >>= 1; ++t;}
    for (int i = 0; i < R; ++i) {
        a = rand() % (n - 1) + 1;
        if (miller_rabin_subroutine(a, n, x, t))
        return false;
    }
    return true;
}
```

4.15 Integer Factorization - Pollard Rho

```
// Pre-condition: n > 1
// Expected Runtime: O(sqrt(p)) where p is a small prime
// factor of n
//
constexpr int MAX_PRIME_FACTORS = 1000;
ll factor[MAX_PRIME_FACTORS];    // Save the result
int tol;    // Count of prime factors

// Base on Brent's implementation
ll pollard_rho(ll n) {
    if (n % 2 == 0) return 2;
    if (n % 3 == 0) return 3;
    ll w = 0, a = 0, val = 1, g;
    ll c = rand() % (n - 1) + 1;
    for (ll k = 2;; k <<= 1, a = w, val = 1) {
        for (ll i = 1; i <= k; ++i) {
            w = (multmod(w, w, n) + c) % n;
            val = multmod(val, abs(w - a), n);
            if (! (i & 127)) {
                g = bingcd(val, n);
                if (g > 1) return g;
            }
        }
        g = bingcd(val, n);
        if (g > 1) return g;
    }
}

void find_prime_factors(ll n) {
    if (miller_rabin(n)) { factor[tol++] = n; return;}
    ll p = n;
    while (p >= n) p = pollard_rho(n);
    find_prime_factors(p); find_prime_factors(n / p);
}

// Usage:
int main() {
    tol = 0;    // reset the counter
    find_prime_factors(1231279381729381);
    // sort the array 'factor' if you need the order
    // after the factorization
    for (int i = 0; i < tol; ++i) cout << factor[i] << ' ';
}
```

4.16 Gaussian Elimination

$O(\min(n,m)nm)$ where $n =$# of equations, $m = \#of variables$.

```
const int MULT = 2;
int gauss (vector < vector<double> >& a, vector<double> & ans)
    int n = (int) a.size();
    int m = (int) a[0].size() - 1;

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

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

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

4.17 (Inverse) Fast Fourier Transformation

$O(n \log n)$ in length of A for both.

```
const double PI = acos(-1.0);
const ll Max = 1e6+10;
ll bound, logBound;
cd root[Max], arrA[Max], arrB[Max];
ll perm[Max];
ll prod[Max];
// Run preCalc beforehand
void fft(cd* arr) {
    for(ll i = 0; i < bound; i++) {
        if(i < perm[i]) {
            swap(arr[i], arr[perm[i]]);
        }
    }
    for(ll len = 1; len < bound; len *= 2) {
        for(ll pos = 0; pos < bound; pos += 2 * len) {
```

```
            for(ll i = 0; i < len; i++) {
                cd x = arr[pos + i];
                cd y = arr[pos+i+len]
                    * root[bound/len/2*i];
                arr[pos + i] = x + y;
                arr[pos + i + len] = x - y;
            }
        }
    }
}

void preCalc() {
    ll hb = -1;
    root[0] = 1;
    double angle = 2 * pi / bound;
    for(ll i = 1; i < bound; i++) {
        if((i & (i - 1)) == 0) hb++;
        root[i] = cd(cos(angle * i), sin(angle * i));
        perm[i] = perm[i ^ (1 << hb)]
            + (1 << (logBound - hb - 1));
    }
}

void IFFT(vector<cd> &A) {
    for (auto &p : A) p = conj(p);
    FFT(A);
    // not needed for poly mult
    for (auto &p : A) p = conj(p));
    for (auto &p : A) p /= A.size();
}
```

4.18 Multiplying Polynomials

$O(n \log n)$ in size of largest polynomial.

```
void mult(vector<ll> &a, vector<ll> &b, vector<ll> &c) {
    logBound = 0;
    while((1<<logBound) < a.size()
      || (1<<logBound) < b.size())
        logBound++;
    logBound++;
    bound = (1<<logBound);
    preCalc();
    for(ll i = 0; i < a.size(); i++) {
        arrA[i] = cd(a[i], 0);
    }
    for(ll i = a.size(); i < bound; i++) {
        arrA[i] = cd(0, 0);
    }
    for(ll i = 0; i < b.size(); i++) {
        arrB[i] = cd(b[i], 0);
    }
    for(ll i = b.size(); i < bound; i++) {
        arrB[i] = cd(0, 0);
    }
    fft(arrA);
    fft(arrB);
    for(ll i = 0; i < bound; i++) {
        arrA[i] *= arrB[i];
    }
    fft(arrA);
    reverse(arrA + 1, arrA + bound);
    c.resize(bound);
    for(ll i = 0; i < bound; i++) {
        arrA[i] /= bound;
        ll temp = (arrA[i].real() > 0 ?
            arrA[i].real()+.5 : arrA[i].real() - .5);
        c[i] = temp;
    }
    while(c.size() && c.back() == 0) c.pop_back();
}
```

5 String Processing

5.1 String Hashing

O(n) in size of string. $\frac{1}{m}$ chance of collision.

```
ll prhash(std::string s, ll g, ll m){
    ll h = 0, p = 1;
    for(int i = 0; i < s.size(); i++){
        h = (h + s[i] * p) % m;
        p = (p * g) % m;}
    return h;}
```

5.2 String Matching - Knuth-Morris-Pratt

O(n) in size of text

```
void kmp(string t, string p, vi &indicies){
    vi b(p.size() + 1);
    ll i = 0, j = -1; b[0] = -1;
    while(i < p.size()){
        while(j >= 0 && p[i] != p[j]) j = b[j];
        i++; j++;
        b[i] = j;}
    i = 0; j = 0;
    while(i < t.size()){
        while(j >= 0 && t[i] != p[j]) j = b[j];
        i++; j++;
        if(j == p.size()){
            indicies.push_back(i);
            j = b[j];}}}
```

5.3 Suffix Array

5.3.1 Construction

$O(n \log n)$ where n is length of string.

```
vector<int> sais(const vector<int> &s) {
    int n = (int) s.size() - 1;
    int mv = *max_element(s.begin(), s.end()) + 1;
    vector<int> SA(n + 1, -1);
    vector<int> bucket(mv), lbucket(mv), sbucket(mv);

    for (auto x : s) ++bucket[x];
    for (int i = 1; i < mv; ++i) {
        bucket[i] += bucket[i - 1];
        lbucket[i] = bucket[i - 1];
        sbucket[i] = bucket[i] - 1;
    }

    vector<bool> t(n + 1);
    t[n] = 1;
    for (int i = n - 1; i >= 0; --i) {
        t[i] = (s[i]<s[i+1]?1:(s[i]>s[i+1]?0:t[i+1]));
    }

    auto is_lms_char = [&](int i) {
        return i > 0 && t[i] == 1 && t[i - 1] == 0;
    };
    auto equal_substring = [&](int x, int y) {
        do {
            if (s[x] != s[y]) return false;
            ++x;
            ++y;
        } while (!is_lms_char(x) && !is_lms_char(y));
        return s[x] == s[y];
    };
    auto induced_sort = [&]() {
        for (int i = 0; i <= n; ++i) {
            if (SA[i] > 0 && t[SA[i] - 1] == 0) {
                SA[lbucket[s[SA[i] - 1]]++] = SA[i] - 1;
            }
        }
        for (int i = 1; i < mv; ++i) {
            sbucket[i] = bucket[i] - 1;
        }
        for (int i = n; i >= 0; --i) {
            if (SA[i] > 0 && t[SA[i] - 1] == 1) {
                SA[sbucket[s[SA[i] - 1]]--] = SA[i] - 1;
            }
        }
    };
```

```
    vector<int> pos;
    for (int i = 1; i <= n; ++i) {
        if (t[i] == 1 && t[i - 1] == 0) {
            pos.emplace_back(i);
        }
    }
    for (auto x : pos) SA[sbucket[s[x]]--] = x;
    induced_sort();
    vector<int> name(n + 1, -1);
    int lx = -1, cnt = 0;
    bool flag = true;
    for (const auto &x : SA) {
        if (is_lms_char(x)) {
            if (lx >= 0 && !equal_substring(lx, x)) {
                ++cnt;
            }
            if (lx >= 0 && cnt == name[lx]) {
                flag = false;
            }
            name[x] = cnt;
            lx = x;
        }
    }
    vector<int> s1;
    for (const auto &x : name) {
        if (x != -1) {
            s1.emplace_back(x);
        }
    }
    vector<int> sa1;
    if (flag) {
        int n1 = s1.size();
        sa1.resize(n1);
        for (int i = 0; i < n1; ++i) sa1[s1[i]] = i;
    } else {
        sa1 = sais(s1);
    }
    lbucket[0] = sbucket[0] = 0;
    for (int i = 1; i < mv; ++i) {
        lbucket[i] = bucket[i - 1];
        sbucket[i] = bucket[i] - 1;
    }

    fill(SA.begin(), SA.end(), -1);
    for (int i = (int) sa1.size() - 1; i >= 0; --i) {
        SA[sbucket[s[pos[sa1[i]]]]--] = pos[sa1[i]];
    }
    induced_sort();
    return SA;
}

string text;
vector<int> sa;
void sa_init() {
    vector<int> s(text.begin(), text.end());
    s.emplace_back(0);
    sa = sais(s);
    sa = vector<int>(sa.begin() + 1, sa.end());
}

vector<int> sa_find(const string &p){
    auto r = equal_range(sa.begin(), sa.end(), -1, [&](int i, int j) {
        int a = 1;
        if (i == -1) {
            swap(i, j);
            a = -1;
        }
        return a * text.compare(i, p.size(), p) < 0;
    });
    vector<int> occ(r.first, r.second);
    return occ;
}
```

5.3.2 String Matching

$O(m \log n)$ where m is the size of the pattern, and n is the size of the text. Better than KMP if searching multiple patterns. Similar to AC automaton in performance.

```
vector<int> sa_find(const string &p){
    auto r = equal_range(sa.begin(), sa.end(), -1,
        [&](int i, int j) {
        int a = 1;
        if (i == -1) {
            swap(i, j);
            a = -1;
        }
        return a * text.compare(i, p.size(), p) < 0;
    });
    vector<int> occ(r.first, r.second);
    return occ;
}
```

5.3.3 Longest Common Prefix

$O(n)$ in size of string

```
vector<int> lcp(string s, vector<int> p) {
    int n = s.size();
    vector<int> rank(n, 0);
    for(int i = 0; i < n; i++) rank[p[i]] = i;
    int k = 0;
    vector<int> lcp(n-1, 0);
    for(int i = 0; i < n; i++) {
        if(rank[i] == n - 1) {
            k = 0;
            continue;
        }
        int j = p[rank[i] + 1];
        while (i+k<n&&j+k<n&&s[i+k]==s[j+k])k++;
        lcp[rank[i]] = k;
        if(k) k--;
    }
    return lcp;
}
```

5.4 Lyndon Factorization - Duval

$O(n)$ in size of string

```
vector<string> duval(string const& s) {
    int n = s.size(), i = 0;
    vector<string> f;
    while (i < n) {
        int j = i + 1, k = i;
        while (j < n && s[k] <= s[j]) {
            if (s[k] < s[j]) k = i;
            else k++;
            j++;}
        while (i <= k) {
            f.push_back(s.substr(i, j - k));
            i += j - k;}}
    return f;}
```

5.5 Minimal Rotation

Get the minimal rotation of the string by Booth's algorithm in $O(|s|)$

```
int booth(string s) {
    s += s;
    int length = (int)s.size();
    vector<int> f(length, -1);
    int k = 0;
    for (int j = 1, i; j < length; ++j) {
        char sj = s.at(j);
        i = f.at(j - k - 1);
        while (i != -1 && sj != s.at(k + i + 1)) {
            if (sj < s.at(k + i + 1)) {
```

```
            k = j - i - 1;
        }
        i = f.at(i);
    }
    if (sj != s.at(k + i + 1)) {
        if (sj < s.at(k)) {
            k = j;
        }
        f.at(j - k) = -1;
    } else {
        f.at(j - k) = i + 1;
    }
}
return k;
}
// Check if two strings are rotated
bool rotateString(string A, string B) {
    if (A.size() != B.size()) return false;
    rotate(A.begin(), A.begin() + booth(A), A.end());
    rotate(B.begin(), B.begin() + booth(B), B.end());
    return A == B;
}
```

6 Geometry

6.1 Points

O(1) for all operations

```
typedef double P_TYPE;
struct point{
    P_TYPE x, y;
    point(){x = y = 0.0;}
    point(P_TYPE _x, P_TYPE _y){x = _x; y = _y;}
    point& operator += (const point o){
        x += o.x; y += o.y;
        return *this;}
    point& operator -= (const point o){
        x -= o.x; y -= o.y;
        return *this;}
    point& operator *= (const P_TYPE o){
        x *= o; y *= o;
        return *this;}
    point& operator /= (const P_TYPE o){
        x /= o; y /= o;
        return *this;}
    point operator + (const point o) const{
        return point(*this) += o;}
    point operator - (const point o) const{
        return point(*this) -= o;}
    point operator * (const P_TYPE o) const{
        return point(*this) *= o;}
    point operator / (const P_TYPE o) const{
        return point(*this) /= o;}
    bool operator < (const point o) const{
        if(fabs(x - o.x) > EPS) return x < o.x;
        else return y < o.y;}
    bool operator == (const point o) const{
        return ((abs(x-o.x)<EPS)&&(abs(y-o.y)<EPS));}
    P_TYPE dist(const point o) const{
        return hypot(x-o.x, y-o.y);}
    P_TYPE dot(const point o) const{
        return x * o.x + y * o.y;}
    P_TYPE cross(const point o) const{
        return x * o.y - y * o.x;}
    P_TYPE norm_squared() const{
        return dot(*this);}
};
bool cw(point a, point b, point c){
    return (b-a).cross(c-a) < 0;}
bool ccw(point a, point b, point c){
    return (b-a).cross(c-a) > 0;}
bool collinear(point a, point b, point c){
    return fabs((b-a).cross(c-a)) < EPS;}
```

6.1.1 Orientation Tests

```
bool cw(point a, point b, point c){
    point q(b.x-a.x, b.y-a.y), r(c.x-a.x,c.y-a.y);
    return q.cross(r) < 0;
}
bool ccw(point a, point b, point c){
    point q(b.x-a.x, b.y-a.y), r(c.x-a.x,c.y-a.y);
    return q.cross(r) > 0;
}
bool collinear(point a, point b, point c){
    point q(b.x-a.x, b.y-a.y), r(c.x-a.x,c.y-a.y);
    return fabs(q.cross(r)) < EPS;
}
```

6.2 Lines

O(1) for all operations

```
struct line{
    double a,b,c;
    line(double _a, double _b, double _c){
        a=_a; b=_b; c=_c;}
    line(point p1, point p2){
        a = p1.y - p2.y; b = p2.x - p1.x;
        c = p1.x * p2.y - p2.x * p1.y;}
    double distToPoint(point p){
        return fabs(a*p.x+b*p.y+c)/sqrt(a*a+b*b);}};
```

6.3 Circles

contains(point) O(1)
contains(vector<point>) O(n) in size of vector

```
struct circle{
    point c;
    double r;
    bool contains(point p){
        return c.dist(p) <= r*M_EPS;
    }
    bool contains(vector<point> p){
        for(auto i : p){
            if(!contains(i)) return 0;
        }
        return 1;
    }
};
```

6.3.1 Angle Conversions

O(1) for all operations

```
inline double radToDeg(double d){return d*180/M_PI;}
inline double degToRad(double r){return r*M_PI/180;}
```

6.3.2 Circumscribed Circle

O(1) for both functions.

```
circle circum_circle(point a, point b) {
        point c = (a + b)/2.0;
        return circle{c, c.dist(a)};
}

circle circum_circle(point a, point b, point c){
    point ba = b - a, ca = c - a;
    double d = 2 * ba.cross(ca);
        double x = (ca.y*ba.dot(ba)-ba.y*ca.dot(ca))/d;
        double y = (ba.x*ca.dot(ca)-ca.x*ba.dot(ba))/d;
        point p(x,y); p += a;
        return {p, p.dist(a)};
}
```

6.3.3 Smallest Circumscribed Circle

Randomized Algorithm. O(n) time with respect to size of poly.

```
circle sc3(vector<point> pts, int e, point p, point q){
    circle circ = circum_circle(p, q);
    circle l  = {{0,0}, -1}, r = {{0,0}, -1};
    point pq = q-p;
    for (int i = 0; i < e; i++) {
        point cur = pts[i];
        if (circ.contains(cur)) continue;
        double cp = pq.cross(cur-p);
        circle c = circum_circle(p, q, cur);
        double pqc = pq.cross(c.c-p);
        if (c.r < 0) continue;
        else if (cp>0&&(l.r<0||pqc>pq.cross(l.c-p)))l=c;
        else if (cp<0&&(r.r<0||pqc<pq.cross(r.c-p)))r=c;
    }
    if (l.r < 0 && r.r < 0) return circ;
    else if (l.r < 0) return r;
    else if (r.r < 0)return l;
    else return l.r <= r.r ? l : r;
}
circle sc2(vector<point> pts, int e, point p) {
    circle c{p, 0};
    for (int i = 0; i < e; i++){
        point q = pts[i];
        if (!c.contains(q)) {
            if (c.r == 0) c = circum_circle(p, q);
            else c = sc3(pts, i + 1, p, q);
        }
    }
    return c;
}s
circle smallest_circle(vector<point> points) {
    shuffle(points.begin(), points.end(), rng);
    circle c = {{0,0}, -1};
    for (int i = 0; i < points.size(); i++) {
        if (c.r < 0 || !c.contains(points[i])){
            c = sc2(points, i + 1, points[i]);
        }
    }
    return c;
}
```

6.4 Triangles

6.4.1 Trigonometry

The Law of Cosines: $c^2 = a^2 + b^2 - 2ab \cdot \cos(\theta)$

6.5 Polygons

6.5.1 Representation

Clockwise order of points; End point is same as first point; _p cannot be empty. O(n) time for all operations

```
struct polygon{
    vector<point> p;
    polygon(vector<point> _p){
        p = _p;
        p.push_back(p[0]);}
    double perimeter(){
        double sum = 0.0;
        for(int i = 0; i < p.size()-1; i++){
            sum +=p[i].dist(p[i+1]);}
        return sum;}
    double area(){
        double product = 0.0, x1, y1, x2, y2;
        for (int i = 0; i < p.size()-1; i++){
            product += p[i].cross(p[i+1]);}
        return fabs(product)/2.0;}};
```

6.5.2 Convex Hull - Graham Scan

O(nlog(n)) where n is the number of points

```
void to_convex_hull(vector<point> &p){
    if(p.size() == 1) return;
    sort(p.begin(), p.end());
    point p1 = p[0], p2 = p.back();
    vector<point> u, d;
    u.push_back(p1); d.push_back(p1);
    for(int i = 1; i < p.size(); i++) {
        if(i == p.size() - 1 || cw(p1, p[i], p2)) {
            while (u.size() >= 2){
                if(!cw(u[u.size()-2], u.back(), p[i]))
                    u.pop_back();
                else break;
            }
            u.push_back(p[i]);
        }
        if(i == p.size() - 1 || ccw(p1, p[i], p2)) {
            while(d.size() >= 2){
                if(!ccw(d[d.size()-2], d.back(), p[i]))
                    d.pop_back();
                else break;
            }
            d.push_back(p[i]);
        }
    }
    p.clear();
    for(int i=0; i<u.size(); i++) p.push_back(u[i]);
    for(int i=d.size()-2; i>0; i--) p.push_back(d[i]);
}
```

6.5.3 Rotating Caliper

ch parameter must be CW or CCW convexhull.
rc_2pt() returns max dist for all pairs in ch in $O(n)$ time.
rc_3pt() returns max area for all triplets in ch in $O(n^2)$ time.

```
double area(point p1, point p2, point p3)
    {return abs((p2-p1).cross(p3-p1))/2.0;}

double rc_2pt(const vector<point> &ch){
    int n = ch.size(), j = 1;
    double d = 0.0;
    for(int i = 0; i < n - 1; i++){
        while(j + 1 < n && ch[i].dist(ch[j])
            < ch[i].dist(ch[j+1])) j++;
        d = max(d, ch[i].dist(ch[j]));
    }
    return d;
}

double rc_3pt(const vector<point> &ch){
    int n = ch.size();
    double a = 0.0;
    for(int i = 0; i < n; i++){
        int k = i + 2;
        for(int j = i + 1; j < n; j++){
            while(k + 1 < n && area(ch[i], ch[j], ch[k])
                < area(ch[i], ch[j], ch[k+1])) k++;
            a = max(a, area(ch[i], ch[j], ch[k]));
        }
    }
    return a;
}
```

7 Miscellaneous

7.1 Read and Write Integers Faster

```
namespace IO {
    // Read until EOF
    template<typename T>
    bool can_read(T &t) {
        int n = 0;
        int ch = getchar();
        while (!isdigit(ch)) {
```

```
            if (ch == EOF) return false;
            n |= ch == '-';
            ch = getchar();
        }
        t = 0;
        while (isdigit(ch)) {
            t = t * 10 + ch - 48;
            ch = getchar();
        }
        if (n) t = -t;
        return true;
    }
    template<typename T, typename... Args>
    bool can_read(T &t, Args &... args) {
        return can_read(t) && can_read(args...);
    }
    template<typename T>
    void read(T &t) {
        int n = 0;
        int c = getchar();
        t = 0;
        while (!isdigit(c))n|=c=='-',c=getchar();
        while (isdigit(c))t=t*10+c-48,c=getchar();
        if (n) t = -t;
    }
    template<typename T, typename... Args>
    void read(T &t, Args &... args) {
        read(t);
        read(args...);
    }
    template<typename T>
    void write(T x) {
        if (x < 0) x = -x, putchar('-');
        if (x > 9) write(x / 10);
        putchar(x % 10 + 48);
    }
    template<typename T>
    void writeln(T x) {
        write(x);
        putchar('\n');
    }
}
```

7.2 Read and Write Int128

```
using int128 = __int128_t;  // Type alias
// Convert int128 to string
string print_int128(int128 a) {
    if (!a) {
        return "0";
    }
    string s;
    while (a) {
        s = char(llabs((long long)(a % 10)) + '0') + s;
        if (a < 0 && a > -10) {
            s = '-' + s;
        }
        a /= 10;
    }
    return s;
}
// Convert string to int128
int128 stoint128(const string &s) {
    int128 a = 0, sgn = 1;
    for (const char &ch : s) {
        if (ch == '-') sgn *= -1;
        else a = a * 10 + sgn * (ch - '0');
    }
    return a;
}
// Usage:
int main() {
    string s = "73786976294838206464";
    int128 p = stoint128(s);
    cout << (int)(p - p) << endl;
    cout << print_int128(p * 2) << endl;
}
```

7.3 Policy Base Data Structure

Include the statements below before using pbds

```
// #include<bits/extc++.h> // For pbds
#include<ext/pb_ds/assoc_container.hpp>
#include<ext/pb_ds/tree_policy.hpp> // For tree
#include<ext/pb_ds/hash_policy.hpp> // For hashtable
#include<ext/pb_ds/trie_policy.hpp> // For trie
using namespace __gnu_pbds;
```

7.3.1 Red Black Tree

Insert/Remove/Get kth/Get succ/Get prev/Get rank in $O(\lg n)$

```
// Type alias for a red-black tree
// whose key is pair<ll, ll>
using RBTree = tree<ii,
null_type,less<>,
rb_tree_tag,tree_order_statistics_node_update>;

RBTree tr;
tr.insert({1, 3}); // insert;
tr.erase({1, 4}); // remove;
tr.order_of_key({1, 0}); // get rank

// get kth, k >= 0, return an iterator
tr.find_by_order(k);

tr.join(tr2);  // merge tr2 into tr

// split tr such that
// nodes > v will be moved to tr2
tr.split(v,tr2);

tr.lower_bound(x); // same as std::set.lower_bound
tr.upper_bound(x); // same as std::set.upper_bound
```

7.3.2 Trie

```
using Trie = trie
<string,null_type,
trie_string_access_traits<>,
pat_trie_tag,
trie_prefix_search_node_update>;

Trie tr; // initialize a trie instance
string s = "abcd";
tr.insert(s); // insert s
tr.erase(s);  // remove s
tr.join(tr2); // merge tr2 into tr, tr2 will be empty

// iterate all strings in the trie
// that starts with "abcd"
auto range=base.prefix_range("abcd");
for(auto it=range.first; it!=range.second; ++it)
    cout << *it << '\n';
```

7.3.3 Hashtable uses probing

```
gp_hash_table<int, bool> h;
// The usage is the same as unordered_map
// And it is faster than unordered_map
// in most cases
```

7.4 Better Hash Function

```
struct myHash {
    static uint64_t splitmix64(uint64_t x) {
        // http://xorshift.di.unimi.it/splitmix64.c
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
```

```cpp
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }

    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM =
                chrono::steady_clock::now().
                time_since_epoch().count();
        return splitmix64(x + FIXED_RANDOM);
    }

    // For a pair of integers
    size_t operator()(pair<uint64_t, uint64_t> x) const {
        static const uint64_t FIXED_RANDOM =
                chrono::steady_clock::now().
                time_since_epoch().count();
        return splitmix64(x.first + FIXED_RANDOM)^
               (splitmix64(x.second + FIXED_RANDOM)>>1);
    }
};
// Usage:
unordered_set<pair<int, int>, myHash> uset;
unordered_map<int, int, myHash> umap;
gp_hash_table<int, null_type, myHash> hashTable;
```

7.5 Hash Function for Fixed Length Array

```cpp
constexpr int ARRAY_SIZE = 26;
template<typename T>
struct myHashFunc {
    std::size_t operator()(const array<T, ARRAY_SIZE> &A)
    const {
        std::size_t h = 0;
        for (const auto &i : A)
            h ^= std::hash<int>{}(i)
                + 0x9e3779b9
                + (h << 6) + (h >> 2);
        return h;
    }
};
// Usage:
unordered_set<array<int, ARRAY_SIZE>,
    myHashFunc<int> > unset;
unordered_map<array<ll, ARRAY_SIZE>,
    int, myHashFunc<ll> > unmap;
```

7.6 Hash Function for a Vector

```cpp
template<typename T>
struct VectorHashFunc {
    std::size_t operator()(const vector<T> &v) const {
        std::size_t h = v.size();
        for(const auto &i : v) {
            h ^= std::hash<T>{}(i)
            + 0x9e3779b9
            + (h << 6) + (h >> 2);
        }
        return h;
    }
};
// Usage:
unordered_set<vector<int>, VectorHashFunc<int> > unset;
```

7.7 #Pragmas Optimization

```cpp
#pragma GCC optimize ("Ofast")
#pragma GCC optimize ("unroll-loops")

// This does work in Codeforces but not in Kattis
#pragma GCC target("sse,sse2,sse3,ssse3,sse4,popcnt,abm,mmx,avx,tune=native")
```

7.8 Bitwise

7.8.1 Next Bit Permutation

```cpp
// Compute the lexicographically next bit permutation
// 0b10001 --> 0b10010
// 0b00000 --> 0b11111 (since 32bits, we have 32 '1's)
unsigned int next_bit_permutation(unsigned current_bit_perm){
    unsigned int v = current_bit_perm;
    unsigned int w;
    unsigned int t = v | (v - 1);
    w = (t + 1) | (((~t & -~t) - 1) >>
        (__builtin_ctz(v) + 1));
    return w;
}
```

7.8.2 Loop Through All Subsets

For example, if $S = 0b10110$, then loop through will obtain 0b10100, 0b10010, 0b10000, 0b00110, 0b00100, 0b00010. The set S and $\emptyset$ will be ignored.

```cpp
int s = 0b10110;
for (int ss=(s-1)&s; ss; ss=(ss-1)&s) {
    cout << bitset<5>(ss).to_string() << '\n';
}
```

7.8.3 Other Tricks

```cpp
namespace Bitwise {
    // All boolean array are in 0-based indexing
    bool isOn(int S, int j) { return (S & (1 << j)); }
    void setBit(int &S, int j) { S |= (1 << j); }
    void clearBit(int &S, int j) { S &= ~(1 << j); }
    void toggleBit(int &S, int j) { S ^= (1 << j); }
    // S&(-S) is 2^j such that the j-th of S is 0
    int lowBit(int S) { return S & (-S); }
    // set S to all '1' bit of length n
    void setAll(int &S, int n) { S = (1<<n)-1; }
    // Obtain S%n such that n is a power of 2
    int modulo(int S, int n) { return S & (n - 1); }
    bool isPowerOfTwo(int S) { return !(S & (S-1)); }
    // pre: the last bit of S has not been turned off
    int turnOffLastBit(int S) { return S & (S-1); }

    // __builtin_ctz(2^n) = n
    // Count how many bits are set in 'x'
    // __builtin_popcount(x)
    // __builtin_popcountl(x)
    // __builtin_ctz(x) return the number of trailing 0
    // __builtin_clz(x) return the number of leading 0

    // Iterate all set bits in a 'bitset' instance
    void f() {
        bitset<10> bs;
        bs[0] = true;
        bs[9] = true;
        bs[5] = true;
        for (size_t i = bs._Find_first(); i < bs.size();
            i = bs._Find_next(i))
            cout << i << ' ';   // 0 5 9
    }
}
```