ACM-ICPC Team Reference Document
IIT Guwahati (KalKeJuwari)

Contents

1 General

1.1 C++ Template

```
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp> //
    gp_hash_table<int, int> == hash map
#include <ext/pb_ds/tree_policy.hpp>
using namespace std;
using namespace __gnu_pbds;
typedef long long ll;
typedef unsigned long long ull;
typedef long double ld;
typedef pair<int, int> pii;
typedef pair<ll, ll> pll;
typedef pair<double, double> pdd;
template <typename T> using min_heap =
    priority_queue<T, vector<T>, greater<T>>;
template <typename T> using max_heap =
    priority_queue<T, vector<T>, less<T>>;
template <typename T> using ordered_set = tree
    <T, null_type, less<T>, rb_tree_tag,
    tree_order_statistics_node_update>;
template <typename K, typename V> using
    hashmap = gp_hash_table<K, V>;
```

```
template<typename A, typename B> ostream&
    operator<<(ostream& out, pair<A, B> p) {
    out << "(" << p.first << ", " << p.second
    << ")"; return out;}
template<typename T> ostream& operator<<(
    ostream& out, vector<T> v) { out << "["; for
    (auto& x : v) out << x << ", "; out << "]";
    return out;}
template<typename T> ostream& operator<<(
    ostream& out, set<T> v) { out << "{"; for(
    auto& x : v) out << x << ", "; out << "}";
    return out; }
template<typename K, typename V> ostream&
    operator<<(ostream& out, map<K, V> m) {
    out << "{"; for(auto& e : m) out << e.first
    << " -> " << e.second << ", "; out << "}";
    return out; }
template<typename K, typename V> ostream&
    operator<<(ostream& out, hashmap<K, V>
    m) { out << "{"; for(auto& e : m) out << e.
    first << " -> " << e.second << ", "; out <<
    "}"; return out; }

#define FAST_IO ios_base::sync_with_stdio(false
    ); cin.tie(NULL)
#define TESTS(t) int NUMBER_OF_TESTS; cin
    >> NUMBER_OF_TESTS; for(int t = 1; t
    <= NUMBER_OF_TESTS; t++)
#define FOR(i, begin, end) for (int i = (begin) - ((
    begin) > (end)); i != (end) - ((begin) > (end))
    ; i += 1 - 2 * ((begin) > (end)))
#define sgn(a) ((a) > eps ? 1 : ((a) < -eps ? -1 : 0)
    )
#define precise(x) fixed << setprecision(x)
#define debug(x) cerr << "> " << #x << " = "
    << x << endl;
#define pb push_back
#define rnd(a, b) (uniform_int_distribution<int
    >((a), (b))(rng))
#ifndef LOCAL
    #define cerr if(0)cout
    #define endl "\n"
#endif
mt19937 rng(chrono::steady_clock::now().
    time_since_epoch().count());
clock_t ___clock___;
void startTime() {___clock___ = clock();}
void timeit(string msg) {cerr << "> " << msg
    << ": " << precise(6) << ld(clock()-
    ___clock___)/CLOCKS_PER_SEC << endl;}
const ld PI = asin(1) * 2;
const ld eps = 1e-14;
const int oo = 2e9;
const ll OO = 2e18;
const ll MOD = 1000000007;
const int MAXN = 1000000;

int main() {
    FAST_IO;
    startTime();

    timeit("Finished");
    return 0;
}
```

1.2 Stress Testing

```
for((i=1;;++i)); do
    echo $i;
    ./gen $i > genIn;
    diff <(./main < genIn) <(./stupid < genIn) ||
        break;
done
```

2 Data Structures

2.1 Disjoin Set Union

```
struct DSU {
    vector<int> par;
    vector<int> sz;

    DSU(int n) {
        FOR(i, 0, n) {
            par.pb(i);
            sz.pb(1);
        }
    }

    int find(int a) {
        return par[a] = par[a] == a ? a : find(par[a
            ]);
    }

    bool same(int a, int b) {
        return find(a) == find(b);
    }

    void unite(int a, int b) {
        a = find(a);
        b = find(b);
        if(sz[a] > sz[b]) swap(a, b);
        sz[b] += sz[a];
        par[a] = b;
    }
};
```

2.2 Fenwick Tree Point Update And Range Query

```
struct Fenwick {
    vector<ll> tree;
    int n;
    Fenwick(){}
```

```
    Fenwick(int _n) {
        n = _n;
        tree = vector<ll>(n+1, 0);
    }
    void add(int i, ll val) { // arr[i] += val
        for(; i <= n; i += i&(-i)) tree[i] += val;
    }
    ll get(int i) { // arr[i]
        return sum(i, i);
    }
    ll sum(int i) { // arr[1]+...+arr[i]
        ll ans = 0;
        for(; i > 0; i -= i&(-i)) ans += tree[i];
        return ans;
    }
    ll sum(int l, int r) {// arr[l]+...+arr[r]
        return sum(r) - sum(l-1);
    }
};
```

2.3 Fenwick Tree Range Update And Point Query

```
struct Fenwick {
    vector<ll> tree;
    vector<ll> arr;
    int n;
    Fenwick(vector<ll> _arr) {
        n = _arr.size();
        arr = _arr;
        tree = vector<ll>(n+2, 0);
    }
    void add(int i, ll val) { // arr[i] += val
        for(; i <= n; i += i&(-i)) tree[i] += val;
    }
    void add(int l, int r, ll val) {// arr[l..r] += val
        add(l, val);
        add(r+1, -val);
    }
    ll get(int i) { // arr[i]
        ll sum = arr[i-1]; // zero based
        for(; i > 0; i -= i&(-i)) sum += tree[i];
        return sum; // zero based
    }
};
```

2.4 Fenwick Tree Range Update And Range Query

```
struct RangedFenwick {
    Fenwick F1, F2; // support range query and
        point update
    RangedFenwick(int _n) {
        F1 = Fenwick(_n+1);
        F2 = Fenwick(_n+1);
    }
    void add(int l, int r, ll v) { // arr[l..r] += v
        F1.add(l, v);
        F1.add(r+1, -v);
        F2.add(l, v*(l-1));
        F2.add(r+1, -v*r);
    }
    ll sum(int i) { // arr[1..i]
        return F1.sum(i)*i-F2.sum(i);
    }
    ll sum(int l, int r) { // arr[l..r]
        return sum(r)-sum(l-1);
    }
};
```

2.5 Fenwick 2D

```
struct Fenwick2D {
    vector<vector<ll>> bit;
    int n, m;
    Fenwick2D(int _n, int _m) {
        n = _n; m = _m;
        bit = vector<vector<ll>>(n+1, vector<ll
            >(m+1, 0));
    }
    ll sum(int x, int y) {
        ll ret = 0;
        for (int i = x; i > 0; i -= i & (-i))
            for (int j = y; j > 0; j -= j & (-j))
                ret += bit[i][j];
        return ret;
    }
    ll sum(int x1, int y1, int x2, int y2) {
        return sum(x2, y2) - sum(x2, y1-1) - sum(
            x1-1, y2) + sum(x1-1, y1-1);
    }
    void add(int x, int y, ll delta) {
        for (int i = x; i <= n; i += i & (-i))
            for (int j = y; j <= m; j += j & (-j))
                bit[i][j] += delta;
    }
};
```

2.6 Segment Tree

```
struct SegmentTree {
    int n;
    vector<ll> t;
    const ll IDENTITY = 0; // OO for min, -OO
        for max, ...
    ll f(ll a, ll b) {
        return a+b;
    }
    SegmentTree(int _n) {
        n = _n; t = vector<ll>(4*n, IDENTITY);
    }
    SegmentTree(vector<ll>& arr) {
        n = arr.size(); t = vector<ll>(4*n,
            IDENTITY);
        build(arr, 1, 0, n-1);
```

```
    }
    void build(vector<ll>& arr, int v, int tl, int tr)
        {
        if(tl == tr) { t[v] = arr[tl]; }
        else {
            int tm = (tl+tr)/2;
            build(arr, 2*v, tl, tm);
            build(arr, 2*v+1, tm+1, tr);
            t[v] = f(t[2*v], t[2*v+1]);
        }
    }
    // sum(1, 0, n-1, l, r)
    ll sum(int v, int tl, int tr, int l, int r) {
        if(l > r) return IDENTITY;
        if (l == tl && r == tr) return t[v];
        int tm = (tl+tr)/2;
        return f(sum(2*v, tl, tm, l, min(r, tm)),
            sum(2*v+1, tm+1, tr, max(l, tm+1),
            r));
    }
    // update(1, 0, n-1, i, v)
    void update(int v, int tl, int tr, int pos, ll
        newVal) {
        if(tl == tr) { t[v] = newVal; }
        else {
            int tm = (tl+tr)/2;
            if(pos <= tm) update(2*v, tl, tm, pos,
                newVal);
            else update(2*v+1, tm+1, tr, pos,
                newVal);
            t[v] = f(t[2*v],t[2*v+1]);
        }
    }
};
```

2.7 Segment Tree With Lazy Propagation

```
// Add to segment, get maximum of segment
struct LazySegTree {
    int n;
    vector<ll> t, lazy;
    LazySegTree(int _n) {
        n = _n; t = vector<ll>(4*n, 0); lazy =
            vector<ll>(4*n, 0);
    }
    LazySegTree(vector<ll>& arr) {
        n = arr.size(); t = vector<ll>(4*n, 0); lazy
            = vector<ll>(4*n, 0);
        build(arr, 1, 0, n-1); // same as in simple
            SegmentTree
    }
    void push(int v) {
        t[v*2] += lazy[v];
        lazy[v*2] += lazy[v];
        t[v*2+1] += lazy[v];
        lazy[v*2+1] += lazy[v];
        lazy[v] = 0;
    }
    void update(int v, int tl, int tr, int l, int r, ll
        addend) {
        if (l > r)
            return;
        if (l == tl && tr == r) {
            t[v] += addend;
            lazy[v] += addend;
        } else {
            push(v);
            int tm = (tl + tr) / 2;
            update(v*2, tl, tm, l, min(r, tm),
                addend);
            update(v*2+1, tm+1, tr, max(l, tm
                +1), r, addend);
            t[v] = max(t[v*2], t[v*2+1]);
        }
    }

    int query(int v, int tl, int tr, int l, int r) {
        if (l > r || r < tl || l > tr) return -OO;
        if (l <= tl && tr <= r) return t[v];
        push(v);
        int tm = (tl + tr) / 2;
        return max(query(v*2, tl, tm, l, r),
                query(v*2+1, tm+1, tr, l, r));
    }
};

// Multiply every element on seg. by 'addend',
    query product of numbers in seg.
struct ProdTree {
    int n;
    vector<ll> t, lazy;
    ProdTree(int _n) {
        n = _n; t = vector<ll>(4*n, 1); lazy =
            vector<ll>(4*n, 1);
    }
    void push(int v, int l, int r) {
        int mid = (l+r)/2;
        t[v*2] = (t[v*2]*pwr(lazy[v], mid-l+1,
            MOD))%MOD;
        lazy[v*2] = (lazy[v*2]*lazy[v])%MOD;
        t[v*2+1] = (t[v*2+1]*pwr(lazy[v], r-(mid
            +1)+1, MOD))%MOD;
        lazy[v*2+1] = (lazy[v*2+1]*lazy[v])%MOD
            ;
        lazy[v] = 1;
    }
    void update(int v, int tl, int tr, int l, int r, ll
        addend) {
        if (l > r)
            return;
        if (l == tl && tr == r) {
            t[v] = (t[v]*pwr(addend, tr-tl+1, MOD
                ))%MOD;
            lazy[v] = (lazy[v]*addend)%MOD;
        } else {
            push(v, tl, tr);
```

```
            int tm = (tl + tr) / 2;
            update(v*2, tl, tm, l, min(r, tm), addend);
            update(v*2+1, tm+1, tr, max(l, tm+1), r, addend);
            t[v] = (t[v*2] * t[v*2+1]) % MOD;
        }
    }

    ll query(int v, int tl, int tr, int l, int r) {
        if (l > r || r < tl || l > tr) return 1;
        if (l <= tl && tr <= r) {
            return t[v];
        }
        push(v, tl, tr);
        int tm = (tl + tr) / 2;
        return (query(v*2, tl, tm, l, min(r, tm)) * query(v*2+1, tm+1, tr, max(l, tm+1), r))%MOD;
    }
};
```

2.8 Treap

```
namespace Treap {
    struct Node {
        Node *l, *r;
        ll key, prio, size;
        Node() {}
        Node(ll key) : key(key), l(nullptr), r(nullptr), size(1) {
            prio = rand() ^ (rand() << 15);
        }
    };

    typedef Node* NodePtr;

    int sz(NodePtr n) {
        return n ? n->size : 0;
    }

    void recalc(NodePtr n) {
        if (!n) return;
        n->size = sz(n->l) + 1 + sz(n->r); // add more operations here as needed
    }

    void split(NodePtr tree, ll key, NodePtr& l, NodePtr& r) {
        if (!tree) {
            l = r = nullptr;
        }
        else if (key < tree->key) {
            split(tree->l, key, l, tree->l);
            r = tree;
        }
        else {
            split(tree->r, key, tree->r, r);
            l = tree;
        }
        recalc(tree);
    }

    void merge(NodePtr& tree, NodePtr l, NodePtr r) {
        if (!l || !r) {
            tree = l ? l : r;
        }
        else if (l->prio > r->prio) {
            merge(l->r, l->r, r);
            tree = l;
        }
        else {
            merge(r->l, l, r->l);
            tree = r;
        }
        recalc(tree);
    }

    void insert(NodePtr& tree, NodePtr node) {
        if (!tree) {
            tree = node;
        }
        else if (node->prio > tree->prio) {
            split(tree, node->key, node->l, node->r);
            tree = node;
        }
        else {
            insert(node->key < tree->key ? tree->l : tree->r, node);
        }
        recalc(tree);
    }

    void erase(NodePtr tree, ll key) {
        if (!tree) return;
        if (tree->key == key) {
            merge(tree, tree->l, tree->r);
        }
        else {
            erase(key < tree->key ? tree->l : tree->r, key);
        }
        recalc(tree);
    }

    void print(NodePtr t, bool newline = true) {
        if (!t) return;
        print(t->l, false);
        cout << t->key << " ";
        print(t->r, false);
        if (newline) cout << endl;
    }
}
```

2.9 Trie

```
struct Trie {
    const int ALPHA = 26;
    const char BASE = 'a';
    vector<vector<int>> nextNode;
    vector<int> mark;
    int nodeCount;
    Trie() {
        nextNode = vector<vector<int>>(MAXN
            , vector<int>(ALPHA, -1));
        mark = vector<int>(MAXN, -1);
        nodeCount = 1;
    }
    void insert(const string& s, int id) {
        int curr = 0;
        FOR(i, 0, (int)s.length()) {
            int c = s[i] - BASE;
            if(nextNode[curr][c] == -1) {
                nextNode[curr][c] = nodeCount
                    ++;
            }
            curr = nextNode[curr][c];
        }
        mark[curr] = id;
    }

    bool exists(const string& s) {
        int curr = 0;
        FOR(i, 0, (int)s.length()) {
            int c = s[i] - BASE;
            if(nextNode[curr][c] == -1) return false
                ;
            curr = nextNode[curr][c];
        }
        return mark[curr] != -1;
    }
};
```

3 Graphs

3.1 Dfs With Timestamps

```
vector<vector<int>> adj;
vector<int> tIn, tOut, color;
int dfs_timer = 0;

void dfs(int v) {
    tIn[v] = dfs_timer++;
    color[v] = 1;
    for (int u : adj[v])
        if (color[u] == 0)
            dfs(u);
    color[v] = 2;
    tOut[v] = dfs_timer++;
}
```

3.2 Lowest Common Ancestor

```
const int MOD = (int)1e9 + 7;
const int LOG = ceil(log2(2e5 + 1));
int gt = 0;
vector<pair<int, int>> times(200001);
vector<bool> visited(200001, false);
vector<vector<int>> adj(200001);
void dfs(int i, int p)
{
    visited[i] = true;
    times[i].first = gt++;
    for (auto it : adj[i])
    {
        if (it != p)
        {
            dfs(it, i);
        }
    }
    times[i].second = gt++;
}
bool ancestor(int i, int j)
{
    return times[i].first <= times[j].first && times[i
        ].second >= times[j].second;
}
signed main()
{

    vector<vector<int>> lifting(n + 1, vector<int
        >(LOG + 1));
    for (int i = 2; i <= n; i++)
    {
        int a;
        cin >> a;
        adj[a].push_back(i);
        lifting[i][0] = a;
    }
    lifting[1][0] = 1;
    dfs(1, -1);
    for (int i = 1; i <= LOG; i++)
        for (int j = 1; j <= n; j++)
            lifting[j][i] = lifting[lifting[j][i - 1]][i - 1];
    // check if already ancestor otherwise
    // for lca of a and b, // lifting[a][0] will be the
        final answer
    for (int i = LOG; i >= 0; i--)
    {
        if (!ancestor(lifting[a][i], b))
        {
            a = lifting[a][i];
        }
    }

    return 0;
}
```

3.3 Strongly Connected Components

```
vector < vector<int> > g, gr; // adjList and
    reversed adjList
vector<bool> used;
vector<int> order, component;

void dfs1 (int v) {
    used[v] = true;
    for (size_t i=0; i<g[v].size(); ++i)
        if (!used[ g[v][i] ])
            dfs1 (g[v][i]);
    order.push_back (v);
}

void dfs2 (int v) {
    used[v] = true;
    component.push_back (v);
    for (size_t i=0; i<gr[v].size(); ++i)
        if (!used[ gr[v][i] ])
            dfs2 (gr[v][i]);
}

int main() {
    int n;
    // read n
    for (;;) {
        int a, b;
        // read edge a -> b
        g[a].push_back (b);
        gr[b].push_back (a);
    }

    used.assign (n, false);
    for (int i=0; i<n; ++i)
        if (!used[i])
            dfs1 (i);
    used.assign (n, false);
    for (int i=0; i<n; ++i) {
        int v = order[n-1-i];
        if (!used[v]) {
            dfs2 (v);
            // do something with the found
                component
            component.clear(); // components are
                generated in toposort-order
        }
    }
}
```

3.4 Bellman Ford Algorithm

```
struct Edge
{
    int a, b, cost;
};

int n, m, v; // v - starting vertex
vector<Edge> e;

/* Finds SSSP with negative edge weights.
 * Possible optimization: check if anything changed
     in a relaxation step. If not - you can break
     early.
 * To find a negative cycle: perform one more
     relaxation step. If anything changes - a
     negative cycle exists.
 */
void solve() {
    vector<int> d (n, oo);
    d[v] = 0;
    for (int i=0; i<n-1; ++i)
        for (int j=0; j<m; ++j)
            if (d[e[j].a] < oo)
                d[e[j].b] = min (d[e[j].b], d[e[j].a] +
                    e[j].cost);
    // display d, for example, on the screen
}
```

3.5 Bipartite Graph

```
class BipartiteGraph
{
private:
    vector<int> _left, _right;
    vector<vector<int>> _adjList;
    vector<int> _matchR, _matchL;
    vector<bool> _used;

    bool _kuhn(int v)
    {
        if (_used[v])
            return false;
        _used[v] = true;
        FOR(i, 0, (int)_adjList[v].size())
        {
            int to = _adjList[v][i] - _left.size();
            if (_matchR[to] == -1 || _kuhn(
                _matchR[to]))
            {
                _matchR[to] = v;
                _matchL[v] = to;
                return true;
            }
        }
        return false;
    }
    void _addReverseEdges()
    {
        FOR(i, 0, (int)_right.size())
        {
            if (_matchR[i] != -1)
            {
                _adjList[_left.size() + i].pb(
                    _matchR[i]);
            }
```

```
        }
    }
    void _dfs(int p)
    {
        if (_used[p])
            return;
        _used[p] = true;
        for (auto x : _adjList[p])
        {
            _dfs(x);
        }
    }
    vector<pii> _buildMM()
    {
        vector<pair<int, int>> res;
        FOR(i, 0, (int)_right.size())
        {
            if (_matchR[i] != -1)
            {
                res.push_back(make_pair(
                    _matchR[i], i));
            }
        }

        return res;
    }

public:
    void addLeft(int x)
    {
        _left.pb(x);
        _adjList.pb({});
        _matchL.pb(-1);
        _used.pb(false);
    }
    void addRight(int x)
    {
        _right.pb(x);
        _adjList.pb({});
        _matchR.pb(-1);
        _used.pb(false);
    }
    void addForwardEdge(int l, int r)
    {
        _adjList[l].pb(r + _left.size());
    }
    void addMatchEdge(int l, int r)
    {
        if (l != -1)
            _matchL[l] = r;
        if (r != -1)
            _matchR[r] = l;
    }
    // Maximum Matching
    vector<pii> mm()
    {
        _matchR = vector<int>(_right.size(), -1);
        _matchL = vector<int>(_left.size(), -1);
        // ^ these two can be deleted if performing
            MM on already partially matched
            graph
        _used = vector<bool>(_left.size() +
            _right.size(), false);

        bool path_found;
        do
        {
            fill(_used.begin(), _used.end(), false);
            path_found = false;
            FOR(i, 0, (int)_left.size())
            {
                if (_matchL[i] < 0 && !_used[i])
                {
                    path_found |= _kuhn(i);
                }
            }
        } while (path_found);

        return _buildMM();
    }

    // Minimum Edge Cover
    // Algo: Find MM, add unmatched vertices
        greedily.
    vector<pii> mec()
    {
        auto ans = mm();
        FOR(i, 0, (int)_left.size())
        {
            if (_matchL[i] != -1)
            {
                for (auto x : _adjList[i])
                {
                    int ridx = x - _left.size();
                    if (_matchR[ridx] == -1)
                    {
                        ans.pb({i, ridx});
                        _matchR[ridx] = i;
                    }
                }
            }
        }
        FOR(i, 0, (int)_left.size())
        {
            if (_matchL[i] == -1 && (int)_adjList
                [i].size() > 0)
            {
                int ridx = _adjList[i][0] - _left.size
                    ();
                _matchL[i] = ridx;
                ans.pb({i, ridx});
            }
        }
        return ans;
    }

    // Minimum Vertex Cover
```

```
    // Algo: Find MM. Run DFS from unmatched
        vertices from the left part.
    // MVC is composed of unvisited LEFT and
        visited RIGHT vertices.
    pair<vector<int>, vector<int>> mvc(bool
        runMM = true)
    {
        if (runMM)
            mm();
        _addReverseEdges();
        fill(_used.begin(), _used.end(), false);
        FOR(i, 0, (int)_left.size())
        {
            if (_matchL[i] == -1)
            {
                _dfs(i);
            }
        }
        vector<int> left, right;
        FOR(i, 0, (int)_left.size())
        {
            if (!_used[i])
                left.pb(i);
        }
        FOR(i, 0, (int)_right.size())
        {
            if (_used[i + (int)_left.size()])
                right.pb(i);
        }
        return {left, right};
    }

    // Maximal Independant Vertex Set
    // Algo: Find complement of MVC.
    pair<vector<int>, vector<int>> mivs(bool
        runMM = true)
    {
        auto m = mvc(runMM);
        vector<bool> containsL(_left.size(), false),
             containsR(_right.size(), false);
        for (auto x : m.first)
            containsL[x] = true;
        for (auto x : m.second)
            containsR[x] = true;
        vector<int> left, right;
        FOR(i, 0, (int)_left.size())
        {
            if (!containsL[i])
                left.pb(i);
        }
        FOR(i, 0, (int)_right.size())
        {
            if (!containsR[i])
                right.pb(i);
        }
        return {left, right};
    }
};
```

3.6 Finding Articulation Points

```
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of
    graph

vector<bool> visited;
vector<int> tin, fup;
int timer;

void processCutpoint(int v) {
    // problem-specific logic goes here
    // it can be called multiple times for the same
        v
}

void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = fup[v] = timer++;
    int children=0;
    for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[to]) {
            fup[v] = min(fup[v], tin[to]);
        } else {
            dfs(to, v);
            fup[v] = min(fup[v], fup[to]);
            if (fup[to] >= tin[v] && p!=-1)
                processCutpoint(v);
            ++children;
        }
    }
    if(p == -1 && children > 1)
        processCutpoint(v);
}

void findCutpoints() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    fup.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs (i);
    }
}
```

3.7 Finding Bridges

```
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of
    graph

vector<bool> visited;
vector<int> tin, fup;
int timer;

void processBridge(int u, int v) {
```

```
    // do something with the found bridge
}

void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = fup[v] = timer++;
    for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[to]) {
            fup[v] = min(fup[v], tin[to]);
        } else {
            dfs(to, v);
            fup[v] = min(fup[v], fup[to]);
            if (fup[to] > tin[v])
                processBridge(v, to);
        }
    }
}

// Doesn't work with multiple edges
// But multiple edges are never bridges, so it's
    easy to check
void findBridges() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    fup.assign(n, -1);
    bridges.clear();
    FOR(i, 0, n) {
        if (!visited[i])
            dfs(i);
    }
}
```

3.8 Max Flow With Ford Fulkerson

```
struct Edge {
    int to, next;
    ll f, c;
    int idx, dir;
    int from;
};

int n, m;
vector<Edge> edges;
vector<int> first;

void addEdge(int a, int b, ll c, int i, int dir) {
    edges.pb({ b, first[a], 0, c, i, dir, a });
    edges.pb({ a, first[b], 0, 0, i, dir, b });
    first[a] = edges.size() - 2;
    first[b] = edges.size() - 1;
}

void init() {
    cin >> n >> m;
    edges.reserve(4 * m);
    first = vector<int>(n, -1);
    FOR(i, 0, m) {
        int a, b, c;
        cin >> a >> b >> c;
        a--; b--;
        addEdge(a, b, c, i, 1);
        addEdge(b, a, c, i, -1);
    }
}

int cur_time = 0;
vector<int> timestamp;

ll dfs(int v, ll flow = OO) {
    if (v == n - 1) return flow;
    timestamp[v] = cur_time;
    for (int e = first[v]; e != -1; e = edges[e].next)
        {
        if (edges[e].f < edges[e].c && timestamp[
            edges[e].to] != cur_time) {
            int pushed = dfs(edges[e].to, min(flow,
                edges[e].c - edges[e].f));
            if (pushed > 0) {
                edges[e].f += pushed;
                edges[e ^ 1].f -= pushed;
                return pushed;
            }
        }
    }
    return 0;
}

ll maxFlow() {
    cur_time = 0;
    timestamp = vector<int>(n, 0);
    ll f = 0, add;
    while (true) {
        cur_time++;
        add = dfs(0);
        if (add > 0) {
            f += add;
        }
        else {
            break;
        }
    }
    return f;
}
```

3.9 Max Flow With Dinic

```
struct Edge {
    int f, c;
    int to;
    pii revIdx;
    int dir;
    int idx;
};
```

```
int n, m;
vector<Edge> adjList[MAX_N];
int level[MAX_N];

void addEdge(int a, int b, int c, int i, int dir) {
    int idx = adjList[a].size();
    int revIdx = adjList[b].size();
    adjList[a].pb({ 0,c,b, {b, revIdx} ,dir,i });
    adjList[b].pb({ 0,0,a, {a, idx} ,dir,i });
}

bool bfs(int s, int t) {
    FOR(i, 0, n) level[i] = -1;
    level[s] = 0;
    queue<int> Q;
    Q.push(s);
    while (!Q.empty()) {
        auto t = Q.front(); Q.pop();
        for (auto x : adjList[t]) {
            if (level[x.to] < 0 && x.f < x.c) {
                level[x.to] = level[t] + 1;
                Q.push(x.to);
            }
        }
    }
    return level[t] >= 0;
}

int send(int u, int f, int t, vector<int>& edgeIdx) {
    if (u == t) return f;
    for (; edgeIdx[u] < adjList[u].size(); edgeIdx[u
        ]++) {
        auto& e = adjList[u][edgeIdx[u]];
        if (level[e.to] == level[u] + 1 && e.f < e.c)
            {
            int curr_flow = min(f, e.c - e.f);
            int next_flow = send(e.to, curr_flow, t
                , edgeIdx);
            if (next_flow > 0) {
                e.f += next_flow;
                adjList[e.revIdx.first][e.revIdx.
                    second].f -= next_flow;
                return next_flow;
            }
        }
    }
    return 0;
}

int maxFlow(int s, int t) {
    int f = 0;
    while (bfs(s, t)) {
        vector<int> edgeIdx(n, 0);
        while (int extra = send(s, oo, t, edgeIdx))
            {
            f += extra;
        }
    }
    return f;
}

void init() {
    cin >> n >> m;
    FOR(i, 0, m) {
        int a, b, c;
        cin >> a >> b >> c;
        a--; b--;
        addEdge(a, b, c, i, 1);
        addEdge(b, a, c, i, -1);
    }
}
```

3.10 Min Cut

```
init();
ll f = maxFlow(); // Ford-Fulkerson
cur_time++;
dfs(0);
set<int> cc;
for (auto e : edges) {
    if (timestamp[e.from] == cur_time &&
        timestamp[e.to] != cur_time) {
        cc.insert(e.idx);
    }
}
// (# of edges in min-cut, capacity of cut)
// [indices of edges forming the cut]
cout << cc.size() << " " << f << endl;
for (auto x : cc) cout << x + 1 << " ";
```

3.11 Number Of Paths Of Fixed Length

Let G be the adjacency matrix of a graph. Then $C_k = G^k$ gives a matrix, in which the value $C_k[i][j]$ gives the number of paths between i and j of length k.

3.12 Shortest Paths Of Fixed Length

Define $A \bigodot B = C \iff C_{ij} = min_{p=1..n}(A_{ip} + B_{pj})$. Let G be the adjacency matrix of a graph. Also, let $L_k = G \bigodot \ldots \bigodot G = G^{\bigodot k}$. Then the value $L_k[i][j]$ denotes the length of the shortest path between i and j which consists of exactly k edges.

3.13 Dijkstra

```
vector<vector<pair<int, int>>> adj;
void dijkstra(int s, vector<int> & d, vector<int>
    & p) {
    int n = adj.size();
    d.assign(n, oo);
    p.assign(n, -1);

    d[s] = 0;
    min_heap<pii> q;
```

```
    q.push({0, s});
    while (!q.empty()) {
        int v = q.top().second;
        int d_v = q.top().first;
        q.pop();
        if (d_v != d[v]) continue;
        for (auto edge : adj[v]) {
            int to = edge.first;
            int len = edge.second;
            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                p[to] = v;
                q.push({d[to], to});
            }
        }
    }
}
```

3.14 Euler Path

```
int n;
vector<vector<int>> g(n, vector<int>(n));
// reading the graph in the adjacency matrix

vector<int> deg(n);
for (int i = 0; i < n; ++i)
{
    for (int j = 0; j < n; ++j)
        deg[i] += g[i][j];
}

int first = 0;
while (first < n && !deg[first])
    ++first;
if (first == n)
{
    cout << -1;
    return 0;
}

int v1 = -1, v2 = -1;
bool bad = false;
for (int i = 0; i < n; ++i)
{
    if (deg[i] & 1)
    {
        if (v1 == -1)
            v1 = i;
        else if (v2 == -1)
            v2 = i;
        else
            bad = true;
    }
}

if (v1 != -1)
    ++g[v1][v2], ++g[v2][v1];

stack<int> st;
st.push(first);
vector<int> res;
while (!st.empty())
{
    int v = st.top();
    int i;
    for (i = 0; i < n; ++i)
        if (g[v][i])
            break;
    if (i == n)
    {
        res.push_back(v);
        st.pop();
    }
    else
    {
        --g[v][i];
        --g[i][v];
        st.push(i);
    }
}

if (v1 != -1)
{
    for (size_t i = 0; i + 1 < res.size(); ++i)
    {
        if ((res[i] == v1 && res[i + 1] == v2) ||
            (res[i] == v2 && res[i + 1] == v1))
        {
            vector<int> res2;
            for (size_t j = i + 1; j < res.size(); ++
                j)
                res2.push_back(res[j]);
            for (size_t j = 1; j <= i; ++j)
                res2.push_back(res[j]);
            res = res2;
            break;
        }
    }
}

for (int i = 0; i < n; ++i)
{
    for (int j = 0; j < n; ++j)
    {
        if (g[i][j])
            bad = true;
    }
}

if (bad)
{
    cout << -1;
}
else
{
    for (int x : res)
        cout << x << " ";
```

```
}
```

3.15 ShivamScc

```
vector<bool> visited; // keeps track of which
    vertices are already visited

// runs depth first search starting at vertex v.
// each visited vertex is appended to the output
    vector when dfs leaves it.
void dfs(int v, vector<vector<int>> const &adj,
    vector<int> &output)
{
    visited[v] = true;
    for (auto u : adj[v])
        if (!visited[u])
            dfs(u, adj, output);
    output.push_back(v);
}

// input: adj -- adjacency list of G
// output: components -- the strongy connected
    components in G
// output: adj_cond -- adjacency list of G^SCC (
    by root vertices)
void strongly_connected_components(vector<
    vector<int>> const &adj,
                                  vector<vector<
                                      int>> &
                                      components
                                      ,
                                  vector<vector<
                                      int>> &
                                      adj_cond)
{
    int n = adj.size();
    components.clear(), adj_cond.clear();

    vector<int> order; // will be a sorted list of G
        's vertices by exit time

    visited.assign(n, false);

    // first series of depth first searches
    for (int i = 0; i < n; i++)
        if (!visited[i])
            dfs(i, adj, order);

    // create adjacency list of G^T
    vector<vector<int>> adj_rev(n);
    for (int v = 0; v < n; v++)
        for (int u : adj[v])
            adj_rev[u].push_back(v);

    visited.assign(n, false);
    reverse(order.begin(), order.end());

    vector<int> roots(n, 0); // gives the root
        vertex of a vertex's SCC

    // second series of depth first searches
    for (auto v : order)
        if (!visited[v])
        {
            std::vector<int> component;
            dfs(v, adj_rev, component);
            components.push_back(component);
            int root = *min_element(begin(
                component), end(component));
            for (auto u : component)
                roots[u] = root;
        }

    // add edges to condensation graph
    adj_cond.assign(n, {});
    for (int v = 0; v < n; v++)
        for (auto u : adj[v])
            if (roots[v] != roots[u])
                adj_cond[roots[v]].push_back(
                    roots[u]);
}
```

4 Geometry

4.1 Line

```
template <typename T>
struct Line { // expressed as two vectors
    Vec<T> start, dir;
    Line() {}
    Line(Vec<T> a, Vec<T> b): start(a), dir(b-a)
        {}

    Vec<ld> intersect(Line l) {
        ld t = ld((l.start-start)^l.dir)/(dir^l.dir);
        // For segment-segment intersection this
            should be in range [0, 1]
        Vec<ld> res(start.x, start.y);
        Vec<ld> dirld(dir.x, dir.y);
        return res + dirld*t;
    }
};
```

4.2 Convex Hull Gift Wrapping

```
vector<Vec<int>> buildConvexHull(vector<Vec<
    int>>& pts) {
    int n = pts.size();
    sort(pts.begin(), pts.end());
    auto currP = pts[0]; // choose some extreme
        point to be on the hull

    vector<Vec<int>> hull;
    set<Vec<int>> used;
```

```
    hull.pb(pts[0]);
    used.insert(pts[0]);
    while(true) {
        auto candidate = pts[0]; // choose some
            point to be a candidate

        auto currDir = candidate-currP;
        vector<Vec<int>> toUpdate;
        FOR(i, 0, n) {
            if(currP == pts[i]) continue;
            // currently we have currP->candidate
            // we need to find point to the left of
                this
            auto possibleNext = pts[i];
            auto nextDir = possibleNext - currP;
            auto cross = currDir ^ nextDir;
            if(candidate == currP || cross > 0) {
                candidate = possibleNext;
                currDir = nextDir;
            } else if(cross == 0 && nextDir.norm
                () > currDir.norm()) {
                candidate = possibleNext;
                currDir = nextDir;
            }
        }
        if(used.find(candidate) != used.end())
            break;
        hull.pb(candidate);
        used.insert(candidate);
        currP = candidate;
    }
    return hull;
}
```

4.3 Convex Hull With Graham's Scan

```
// Takes in >= 3 points
// Returns convex hull in clockwise order
// Ignores points on the border
vector<Vec<int>> buildConvexHull(vector<Vec<
    int>> pts) {
    if(pts.size() <= 3) return pts;
    sort(pts.begin(), pts.end());
    stack<Vec<int>> hull;
    hull.push(pts[0]);
    auto p = pts[0];
    sort(pts.begin()+1, pts.end(), [&](Vec<int> a,
        Vec<int> b) -> bool {
        // p->a->b is a ccw turn
        int turn = sgn((a-p)^(b-a));
        //if(turn == 0) return (a-p).norm() > (b-
            p).norm();
        // ^ among collinear points, take the
            farthest one
        return turn == 1;
    });
    hull.push(pts[1]);
    FOR(i, 2, (int)pts.size()) {
        auto c = pts[i];
        if(c == hull.top()) continue;
        while(true) {
            auto a = hull.top(); hull.pop();
            auto b = hull.top();
            auto ba = a-b;
            auto ac = c-a;
            if((ba^ac) > 0) {
                hull.push(a);
                break;
            } else if((ba^ac) == 0) {
                if(ba*ac < 0) c = a;
                // ^ c is between b and a, so it
                    shouldn't be added to the hull
                break;
            }
        }
        hull.push(c);
    }
    vector<Vec<int>> hullPts;
    while(!hull.empty()) {
        hullPts.pb(hull.top());
        hull.pop();
    }
    return hullPts;
}
```

4.4 Circle Line Intersection

```
double r, a, b, c; // ax+by+c=0, radius is at (0, 0)
// If the center is not at (0, 0), fix the constant c
    to translate everything so that center is at (0,
    0)
double x0 = -a*c/(a*a+b*b), y0 = -b*c/(a*a+b*b
    );
if (c*c > r*r*(a*a+b*b)+eps)
    puts ("no points");
else if (abs (c*c - r*r*(a*a+b*b)) < eps) {
    puts ("1 point");
    cout << x0 << ' ' << y0 << '\n';
}
else {
    double d = r*r - c*c/(a*a+b*b);
    double mult = sqrt (d / (a*a+b*b));
    double ax, ay, bx, by;
    ax = x0 + b * mult;
    bx = x0 - b * mult;
    ay = y0 - a * mult;
    by = y0 + a * mult;
    puts ("2 points");
    cout << ax << ' ' << ay << '\n' << bx <<
        ' ' << by << '\n';
}
```

4.5 Circle Circle Intersection

Let's say that the first circle is centered at $(0, 0)$ (if it's not, we can move the origin to the center of the first circle and adjust the coordinates), and the second one is at (x_2, y_2). Then, let's construct a line $Ax + By + C = 0$, where $A = -2x_2, B = -2y_2, C = x_2^2 + y_2^2 + r_1^2 - r_2^2$. Finding the intersection between this line and the first circle will give us the answer. The only tricky case: if both circles are centered at the same point. We handle this case separately.

4.6 Common Tangents To Two Circles

```
struct pt {
    double x, y;

    pt operator- (pt p) {
        pt res = { x-p.x, y-p.y };
        return res;
    }
};
struct circle : pt {
    double r;
};
struct line {
    double a, b, c;
};
void tangents (pt c, double r1, double r2, vector<line> & ans) {
    double r = r2 - r1;
    double z = sqr(c.x) + sqr(c.y);
    double d = z - sqr(r);
    if (d < -eps) return;
    d = sqrt (abs (d));
    line l;
    l.a = (c.x * r + c.y * d) / z;
    l.b = (c.y * r - c.x * d) / z;
    l.c = r1;
    ans.push_back (l);
}
vector<line> tangents (circle a, circle b) {
    vector<line> ans;
    for (int i=-1; i<=1; i+=2)
        for (int j=-1; j<=1; j+=2)
            tangents (b-a, a.r*i, b.r*j, ans);
    for (size_t i=0; i<ans.size(); ++i)
        ans[i].c -= ans[i].a * a.x + ans[i].b * a.y;
    return ans;
}
```

4.7 Number Of Lattice Points On Segment

Let's say we have a line segment from (x_1, y_1) to (x_2, y_2). Then, the number of lattice points on this segment is given by

$$gcd(x_2 - x_1, y_2 - y_1) + 1.$$

4.8 Pick's Theorem

We are given a lattice polygon with non-zero area. Let's denote its area by S, the number of points with integer coordinates lying strictly inside the polygon by I and the number of points lying on the sides of the polygon by B. Then:

$$S = I + \frac{B}{2} - 1.$$

4.9 Misc

Distance from point to line.

We have a line $l(t) = \vec{a} + \vec{b}t$ and a point $\vec{p}$. The distance from this point to the line can be calculated by expressing the area of a triangle in two different ways. The final formula: $d = \frac{(\vec{p}-\vec{a})\times(\vec{p}-\vec{b})}{|\vec{b}-\vec{a}|}$

Point in polygon.

Send a ray (half-infinite line) from the points to an arbitrary direction and calculate the number of times it touches the boundary of the polygon. If the number is odd, the point is inside the polygon, otherwise it's outside.

Using cross product to test rotation direction.

Let's say we have vectors $\vec{a}$, $\vec{b}$ and $\vec{c}$. Let's define $\vec{ab} = b - a, \vec{bc} = c - b$ and $s = sgn(\vec{ab} \times \vec{bc})$. If $s = 0$, the three points are collinear. If $s = 1$, then $\vec{bc}$ turns in the counterclockwise direction compared to the direction of $\vec{ab}$. Otherwise it turns in the clockwise direction.

Line segment intersection.

The problem: to check if line segments ab and cd intersect. There are three cases:

1. **The line segments are on the same line.** Use cross products and check if they're zero - this will tell if all points are on the same line. If so, sort the points and check if their intersection is non-empty. If it is non-empty, there are an infinite number of intersection points.
2. **The line segments have a common vertex.** Four possibilities: $a = c, a = d, b = c, b = d$.
3. **There is exactly one intersection point that is not an endpoint.** Use cross product to check if points c and d are on different sides of the line going through a and b and if the points a and b are on different sides of the line going through c and d.

Angle between vectors.

$arccos(\frac{\vec{a}\cdot\vec{b}}{|\vec{a}|\cdot|\vec{b}|})$.

Dot product properties.

If the dot product of two vectors is zero, the vectors are orthogonal. If it is positive, the angle is acute. Otherwise it is obtuse.

Lines with line equation.

Any line can be described by an equation $ax + by + c = 0$.

- Construct a line using two points A and B:
  1. Take vector from A to B and rotate it 90 degrees ($(x, y) \to (-y, x)$). This will be (a, b).
  2. Normalize this vector. Then put A (or B) into the equation and solve for c.
- Distance from point to line: put point coordinates into line equation and take absolute value. If (a, b) is not normalized, you still need to divide by $\sqrt{a^2 + b^2}$.
- Distance between two parallel lines: $|c_1 - c_2|$ (if they are not normalized, you still need to divide by $\sqrt{a^2 + b^2}$).
- Project a point onto a line: compute signed distance d between line L and point P. Answer is $P - d(\vec{a, b})$.
- Build a line parallel to a given one and passing through a given point: compute the signed distance d between line and point. Answer is $ax + by + (c - d) = 0$.
- Intersect two lines: $d = a_1b_2 - a_2b_1, x = \frac{c_2b_1-c_1b_2}{d}, y = \frac{c_1a_2-c_2a_1}{d}$. If $abs(d) < \epsilon$, then the lines are parallel.

Half-planes.

Definition: define as line, assume a point (x, y) belongs to half plane iff $ax + by + c \geq 0$.

Intersecting with a convex polygon:

1. Start at any point and move along the polygon's traversal.
2. Alternate points and segments between consecutive points.
3. If point belongs to half-plane, add it to the answer.
4. If segment intersects the half-plane's line, add it to the answer.

Some more techniques.

- Check if point A lies on segment BC:
  1. Compute point-line distance and check if it is 0 (abs less than ϵ).
  2. $\vec{BA}\cdot\vec{BC} \geq 0$ and $\vec{CA}\cdot\vec{CB} \geq 0$.
- Compute distance between line segment and point: project point onto line formed by the segment. If this point is on the segment, then the distance between it and original point is the answer. Otherwise, take minimum of distance between point and segment endpoints.

5 Math

5.1 Linear Sieve

```
ll minDiv[MAXN+1];
vector<ll> primes;

void sieve(ll n){
    FOR(k, 2, n+1){
        minDiv[k] = k;
    }
    FOR(k, 2, n+1) {
        if(minDiv[k] == k) {
            primes.pb(k);
        }
        for(auto p : primes) {
            if(p > minDiv[k]) break;
            if(p*k > n) break;
            minDiv[p*k] = p;
        }
    }
}
```

5.2 Extended Euclidean Algorithm

```
// ax+by=gcd(a,b)
void solveEq(ll a, ll b, ll& x, ll& y, ll& g) {
    if(b==0) {
        x = 1;
        y = 0;
        g = a;
        return;
    }
    ll xx, yy;
    solveEq(b, a%b, xx, yy, g);
    x = yy;
    y = xx-yy*(a/b);
}
// ax+by=c
bool solveEq(ll a, ll b, ll c, ll& x, ll& y, ll& g) {
    solveEq(a, b, x, y, g);
    if(c%g != 0) return false;
    x *= c/g; y *= c/g;
    return true;
}
// Finds a solution (x, y) so that x >= 0 and x is
    minimal
bool solveEqNonNegX(ll a, ll b, ll c, ll& x, ll &y, ll
    & g) {
    if(!solveEq(a, b, c, x, y, g)) return false;
    ll k = x*g/b;
    x = x - k*b/g;
    y = y + k*a/g;
```

```
    if(x < 0) {
        x += b/g;
        y -= a/g;
    }
    return true;
}
```

All other solutions can be found like this:

$$x' = x - k\frac{b}{g}, y' = y + k\frac{a}{g}, k \in \mathbb{Z}$$

5.3 Chinese Remainder Theorem

Let's say we have some numbers m_i, which are all mutually coprime. Also, let $M = \prod_i m_i$. Then the system of congruences

$$\begin{cases} x \equiv a_1 \pmod{m_1} \\ x \equiv a_2 \pmod{m_2} \\ \ldots \\ x \equiv a_k \pmod{m_k} \end{cases}$$

is equivalent to $x \equiv A \pmod{M}$ and there exists a unique number A satisfying $0 \leq A \leq M$.

Solution for two: $x \equiv a_1 \pmod{m_1}, x \equiv a_2 \pmod{m_2}$. Let $x = a_1 + km_1$. Substituting into the second congruence: $km_1 \equiv a_2 - a_1 \pmod{m_2}$. Then, $k = (m_1)^{-1}_{m_2}(a_2 - a_1) \pmod{m_2}$. and we can easily find x. This can be extended to multiple equations by solving them one-by-one.

If the moduli are not coprime, solve the system $y \equiv 0 \pmod{\frac{m_1}{g}}, y \equiv \frac{a_2-a_1}{g} \pmod{\frac{m_2}{g}}$ for y. Then let $x \equiv gy + a_1 \pmod{\frac{m_1 m_2}{g}}$.

5.4 Euler Totient Function

```
// Number of numbers x < n so that gcd(x, n) = 1
ll phi(ll n) {
    if(n == 1) return 1;
    auto f = factorize(n);
    ll res = n;
    for(auto p : f) {
        res = res - res/p.first;
    }
    return res;
}

void phi_1_to_n(int n) {
    vector<int> phi(n + 1);
    for (int i = 0; i <= n; i++)
        phi[i] = i;

    for (int i = 2; i <= n; i++) {
        if (phi[i] == i) {
            for (int j = i; j <= n; j += i)
                phi[j] -= phi[j] / i;
        }
    }
}
```

5.5 Factorization With Sieve

```
// Use linear sieve to calculate minDiv
vector<pll> factorize(ll x) {
    vector<pll> res;
    ll prev = -1;
    ll cnt = 0;
    while(x != 1) {
        ll d = minDiv[x];
        if(d == prev) {
            cnt++;
        } else {
            if(prev != -1) res.pb({prev, cnt});
            prev = d;
            cnt = 1;
        }
        x /= d;
    }
    res.pb({prev, cnt});
    return res;
}
```

5.6 Modular Inverse

```
bool invWithEuclid(ll a, ll m, ll& aInv) {
    ll x, y, g;
    if(!solveEqNonNegX(a, m, 1, x, y, g)) return
        false;
    aInv = x;
    return true;
}
// Works only if m is prime
ll invFermat(ll a, ll m) {
    return pwr(a, m-2, m);
}
// Works only if gcd(a, m) = 1
ll invEuler(ll a, ll m) {
    return pwr(a, phi(m)-1, m);
}
```

5.7 Simpson Integration

```
const int N = 1000 * 1000; // number of steps (
    already multiplied by 2)

double simpsonIntegration(double a, double b){
    double h = (b - a) / N;
    double s = f(a) + f(b); // a = x_0 and b =
        x_2n
    for (int i = 1; i <= N - 1; ++i) {
        double x = a + h * i;
        s += f(x) * ((i & 1) ? 4 : 2);
    }
    s *= h / 3;
    return s;
}
```

5.8 Burnside's Lemma

Let G be a finite group that acts on a set X. For each g in G let X^g denote the set of elements in X that are fixed by g. Burnside's lemma asserts the following formula for the number of orbits:

$$|X/G| = \frac{1}{|G|}\sum_{g\in G}|X^g|.$$

Example. Coloring a cube with three colors.

Let X be the set of 3^6 possible face color combinations. Let's count the sizes of the fixed sets for each of the 24 rotations:

- one 0-degree rotation which leaves all 3^6 elements of X unchanged
- six 90-degree face rotations, each of which leaves 3^3 elements of X unchanged
- three 180-degree face rotation, each of which leaves 3^4 elements of X unchanged
- eight 120-degree vertex rotations, each of which leaves 3^2 elements of X unchanged
- six 180-degree edge rotations, each of which leaves 3^3 elements of X unchanged

The average is then $\frac{1}{24}(3^6 + 6\cdot 3^3 + 3\cdot 3^4 + 8\cdot 3^2 + 6\cdot 3^3) = 57$. For n colors: $\frac{1}{24}(n^6 + 3n^4 + 12n^3 + 8n^2)$.

Example. Coloring a circular stripe of n cells with two colors.

X is the set of all colored striped (it has 2^n elements), G is the group of rotations (n elements - by 0 cells, by 1 cell, ..., by $(n-1)$ cells). Let's fix some K and find the number of stripes that are fixed by the rotation by K cells. If a stripe becomes itself after rotation by K cells, then its 1st cell must have the same color as its $(1 + K \mod n)$-th cell, which is in turn the same as its $(1 + 2K \mod n)$-th cell, etc., until $mK \mod n = 0$. This will happen when $m = n/gcd(K, n)$. Therefore, we have $n/gcd(K, n)$ cells that must all be of the same color. The same will happen when starting from the second cell and so on. Therefore, all cells are separated into $gcd(K, n)$ groups, with each group being of one color, and that yields $2^{gcd(K,n)}$ choices. That's why the answer to the original problem is $\frac{1}{n}\sum_{k=0}^{n-1} 2^{gcd(k,n)}$.

5.9 FFT

```
namespace FFT {
    int n;
    vector<int> r;
    vector<complex<ld>> omega;
    int logN, pwrN;

    void initLogN() {
        logN = 0;
        pwrN = 1;
        while (pwrN < n) {
            pwrN *= 2;
            logN++;
        }
        n = pwrN;
    }

    void initOmega() {
        FOR(i, 0, pwrN) {
            omega[i] = { cos(2 * i*PI / n), sin(2 *
                i*PI / n) };
        }
    }

    void initR() {
        r[0] = 0;
        FOR(i, 1, pwrN) {
            r[i] = r[i / 2] / 2 + ((i & 1) << (logN
                - 1));
        }
    }

    void initArrays() {
        r.clear();
        r.resize(pwrN);
        omega.clear();
        omega.resize(pwrN);
    }

    void init(int n) {
        FFT::n = n;
        initLogN();
        initArrays();
        initOmega();
        initR();
    }

    void fft(complex<ld> a[], complex<ld> f[]) {
        FOR(i, 0, pwrN) {
            f[i] = a[r[i]];
        }
        for (ll k = 1; k < pwrN; k *= 2) {
            for (ll i = 0; i < pwrN; i += 2 * k) {
                for (ll j = 0; j < k; j++) {
                    auto z = omega[j*n / (2 * k)]
                        * f[i + j + k];
                    f[i + j + k] = f[i + j] - z;
                    f[i + j] += z;
                }
            }
        }
    }
}
```

5.10 Gaussian Elimination

```
// The last column of a is the right-hand side of
    the system.
// Returns 0, 1 or oo - the number of solutions.
```

```
// If at least one solution is found, it will be in ans
int gauss (vector < vector<ld> > a, vector<ld> &
    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) {
                ld 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) {
        ld 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 oo;
    return 1;
}
```

5.11 Sprague Grundy Theorem

We have a game which fulfills the following requirements:

- There are two players who move alternately.
- The game consists of states, and the possible moves in a state do not depend on whose turn it is.
- The game ends when a player cannot make a move.
- The game surely ends sooner or later.
- The players have complete information about the states and allowed moves, and there is no randomness in the game.

Grundy Numbers. The idea is to calculate Grundy numbers for each game state. It is calculated like so: $mex(\{g_1, g_2, ..., g_n\})$, where $g_1, g_2, ..., g_n$ are the Grundy numbers of the states which are reachable from the current state. mex gives the smallest nonnegative number that is not in the set ($mex(\{0, 1, 3\}) = 2, mex(\emptyset) = 0$). If the Grundy number of a state is 0, then this state is a losing state. Otherwise it's a winning state.

Grundy's Game. Sometimes a move in a game divides the game into subgames that are independent of each other. In this case, the Grundy number of a game state is $mex(\{g_1, g_2, ..., g_n\}), g_k = a_{k,1} \oplus a_{k,2} \oplus ... \oplus a_{k,m}$ meaning that move k divides the game into m subgames whose Grundy numbers are $a_{i,j}$.

Example. We have a heap with n sticks. On each turn, the player chooses a heap and divides it into two nonempty heaps such that the heaps are of different size. The player who makes the last move wins the game. Let $g(n)$ denote the Grundy number of a heap of size n. The Grundy number can be calculated by going though all possible ways to divide the heap into two parts. E.g. $g(8) = mex(\{g(1) \oplus g(7), g(2) \oplus g(6), g(3) \oplus g(5)\})$. Base case: $g(1) = g(2) = 0$, because these are losing states.

5.12 Binary Power

```
ll power(ll a, ll b, ll m)
{
    if (b == 0)
    {
        return 1;
    }
    ll pr = power(a, b / 2, m);
    if (b % 2)
    {
        return (((pr * pr) % m) * a) % m;
    }
    else
    {
        return (pr * pr) % m;
    }
}
```

5.13 Formulas

$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$; $\sum_{i=1}^{n} i^2 = \frac{n(2n+1)(n+1)}{6}$; $\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$; $\sum_{i=1}^{n} i^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$; $\sum_{i=a}^{b} c^i = \frac{c^{b+1}-c^a}{c-1}, c \neq 1$; $\sum_{i=1}^{n} a_1 + (i-1)d = \frac{n(a_1+a_n)}{2}$; $\sum_{i=1}^{n} a_1 r^{i-1} = \frac{a_1(1-r^n)}{1-r}, r \neq 1$; $\sum_{i=1}^{\infty} ar^{i-1} = \frac{a_1}{1-r}, |r| \leq 1$.

6 Strings

6.1 Hashing

```
struct HashedString {
    const ll A1 = 999999929, B1 = 1000000009, A2
        = 1000000087, B2 = 1000000097;
    vector<ll> A1pwrs, A2pwrs;
    vector<pll> prefixHash;
    HashedString(const string& _s) {
        init(_s);
        calcHashes(_s);
    }
    void init(const string& s) {
        ll a1 = 1;
        ll a2 = 1;
        FOR(i, 0, (int)s.length()+1) {
            A1pwrs.pb(a1);
            A2pwrs.pb(a2);
            a1 = (a1*A1)%B1;
            a2 = (a2*A2)%B2;
        }
    }
    void calcHashes(const string& s) {
        pll h = {0, 0};
        prefixHash.pb(h);
        for(char c : s) {
            ll h1 = (prefixHash.back().first*A1 + c
                )%B1;
            ll h2 = (prefixHash.back().second*A2
                + c)%B2;
            prefixHash.pb({h1, h2});
        }
    }
    pll getHash(int l, int r) {
        ll h1 = (prefixHash[r+1].first - prefixHash[l
            ].first*A1pwrs[r+1-l]) % B1;
        ll h2 = (prefixHash[r+1].second -
            prefixHash[l].second*A2pwrs[r+1-l]) %
             B2;
        if(h1 < 0) h1 += B1;
        if(h2 < 0) h2 += B2;
        return {h1, h2};
    }
};
```

6.2 Prefix Function

```
// pi[i] is the length of the longest proper prefix of
    the substring s[0..i] which is also a suffix
// of this substring
vector<int> prefixFunction(const string& s) {
    int n = (int)s.length();
    vector<int> pi(n);
    for (int i = 1; i < n; i++) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j])
            j++;
        pi[i] = j;
    }
    return pi;
}
```

6.3 Prefix Function Automaton

```
// aut[oldPi][c] = newPi
vector<vector<int>> computeAutomaton(string s)
    {
    const char BASE = 'a';
    s += "#";
    int n = s.size();
    vector<int> pi = prefixFunction(s);
    vector<vector<int>> aut(n, vector<int>(26));
    for (int i = 0; i < n; i++) {
        for (int c = 0; c < 26; c++) {
            if (i > 0 && BASE + c != s[i])
                aut[i][c] = aut[pi[i-1]][c];
            else
                aut[i][c] = i + (BASE + c == s[i])
                    ;
        }
    }
    return aut;
}
vector<int> findOccurs(const string& s, const
    string& t) {
    auto aut = computeAutomaton(s);
    int curr = 0;
    vector<int> occurs;
    FOR(i, 0, (int)t.length()) {
        int c = t[i]-'a';
        curr = aut[curr][c];
        if(curr == (int)s.length()) {
            occurs.pb(i-s.length()+1);
        }
    }
    return occurs;
}
```

6.4 KMP

```
// Knuth-Morris-Pratt algorithm
vector<int> findOccurences(const string& s, const
    string& t) {
    int n = s.length();
    int m = t.length();
    string S = s + "#" + t;
    auto pi = prefixFunction(S);
    vector<int> ans;
    FOR(i, n+1, n+m+1) {
        if(pi[i] == n) {
            ans.pb(i-2*n);
        }
    }
```

```
    return ans;
}
```

6.5 Suffix Array

```
vector<int> sortCyclicShifts(string const& s) {
    int n = s.size();
    const int alphabet = 256; // we assume to use
        the whole ASCII range
    vector<int> p(n), c(n), cnt(max(alphabet, n),
        0);
    for (int i = 0; i < n; i++)
        cnt[s[i]]++;
    for (int i = 1; i < alphabet; i++)
        cnt[i] += cnt[i-1];
    for (int i = 0; i < n; i++)
        p[--cnt[s[i]]] = i;
    c[p[0]] = 0;
    int classes = 1;
    for (int i = 1; i < n; i++) {
        if (s[p[i]] != s[p[i-1]])
            classes++;
        c[p[i]] = classes - 1;
    }
    vector<int> pn(n), cn(n);
    for (int h = 0; (1 << h) < n; ++h) {
        for (int i = 0; i < n; i++) {
            pn[i] = p[i] - (1 << h);
            if (pn[i] < 0)
                pn[i] += n;
        }
        fill(cnt.begin(), cnt.begin() + classes, 0);
        for (int i = 0; i < n; i++)
            cnt[c[pn[i]]]++;
        for (int i = 1; i < classes; i++)
            cnt[i] += cnt[i-1];
        for (int i = n-1; i >= 0; i--)
            p[--cnt[c[pn[i]]]] = pn[i];
        cn[p[0]] = 0;
        classes = 1;
        for (int i = 1; i < n; i++) {
            pair<int, int> cur = {c[p[i]], c[(p[i] +
                (1 << h)) % n]};
            pair<int, int> prev = {c[p[i-1]], c[(p[i
                -1] + (1 << h)) % n]};
            if (cur != prev)
                ++classes;
            cn[p[i]] = classes - 1;
        }
        c.swap(cn);
    }
    return p;
}
vector<int> constructSuffixArray(string s) {
    s += "$"; // <- this must be smaller than any
        character in s
    vector<int> sorted_shifts = sortCyclicShifts(s
        );
    sorted_shifts.erase(sorted_shifts.begin());
    return sorted_shifts;
}
```

6.6 Z Algorithm

```
vector<int> z_function(string &s)
{
    int n = s.size();
    vector<int> z(n);
    int l = 0, r = 0;
    for (int i = 1; i < n; i++)
    {
        if (i < r)
        {
            z[i] = min(r - i, z[i - l]);
        }
        while (i + z[i] < n && s[z[i]] == s[i + z[i]])
        {
            z[i]++;
        }
        if (i + z[i] > r)
        {
            l = i;
            r = i + z[i];
        }
    }
    if(n) z[0] = n;
    return z;
}
```

7 Dynamic Programming

7.1 Convex Hull Trick

```
/*
Let's say we have a relation:
dp[i] = min(dp[j] + h[j+1]*w[i]) for j<=i
Let's set k_j = h[j+1], x = w[i], b_j = dp[j]. We
    get:
dp[i] = min(b_j+k_j*x) for j<=i.
This is the same as finding a minimum point on a
    set of lines.
After calculating the value, we add a new line with
k_i = h[i+1] and b_i = dp[i].
*/
struct Line {
    int k;
    int b;

    int eval(int x) {
        return k*x+b;
    }

    int intX(Line& other) {
        int x = b-other.b;
        int y = other.k-k;
```

```
        int res = x/y;
        if(x%y != 0) res++;
        return res;
    }
};

struct BagOfLines {
    vector<pair<Line, int>> lines;

    void addLine(int k, int b) {
        Line current = {k, b};
        if(lines.empty()) {
            lines.pb({current, -OO});
            return;
        }
        int x = -OO;
        while(true) {
            auto line = lines.back().first;
            int from = lines.back().second;
            x = line.intX(current);
            if(x > from) break;
            lines.pop_back();
        }
        lines.pb({current, x});
    }

    int findMin(int x) {
        int lo = 0, hi = (int)lines.size()-1;
        while(lo < hi) {
            int mid = (lo+hi+1)/2;
            if(lines[mid].second <= x) {
                lo = mid;
            } else {
                hi = mid-1;
            }
        }
        return lines[lo].first.eval(x);
    }
};
```

7.2 Divide And Conquer

```
/*
Let A[i][j] be the optimal answer for using i objects
    to satisfy j first
requirements.
The recurrence is:
A[i][j] = min(A[i-1][k] + f(i, j, k)) where f is some
    function that denotes the
cost of satisfying requirements from k+1 to j using
    the i-th object.
Consider the recursive function calc(i, jmin, jmax,
    kmin, kmax), that calculates
all A[i][j] for all j in [jmin, jmax] and a given i
    using known A[i-1][*].
*/
void calc(int i, int jmin, int jmax, int kmin, int
    kmax) {
    if(jmin > jmax) return;
    int jmid = (jmin+jmax)/2;
    // calculate A[i][jmid] naively (for k in kmin...
        min(jmid, kmax){...})
    // let kmid be the optimal k in [kmin, kmax]
    calc(i, jmin, jmid-1, kmin, kmid);
    calc(i, jmid+1, jmax, kmid, kmax);
}

int main() {
    // set initial dp values
    FOR(i, start, k+1){
        calc(i, 0, n-1, 0, n-1);
    }
    cout << dp[k][n-1];
}
```

7.3 Optimizations

1. Convex Hull 1:
   - Recurrence: $dp[i] = \min_{j<i}\{dp[j] + b[j] \cdot a[i]\}$
   - Condition: $b[j] \geq b[j+1], a[i] \leq a[i+1]$
   - Complexity: $\mathcal{O}(n^2) \rightarrow \mathcal{O}(n)$
2. Convex Hull 2:
   - Recurrence: $dp[i][j] = \min_{k<j}\{dp[i-1][k] + b[k] \cdot a[j]\}$
   - Condition: $b[k] \geq b[k+1], a[j] \leq a[j+1]$
   - Complexity: $\mathcal{O}(kn^2) \rightarrow \mathcal{O}(kn)$
3. Divide and Conquer:
   - Recurrence: $dp[i][j] = \min_{k<j}\{dp[i-1][k] + C[k][j]\}$
   - Condition: $A[i][j] \leq A[i][j+1]$
   - Complexity: $\mathcal{O}(kn^2) \rightarrow \mathcal{O}(kn\log(n))$
4. Knuth:
   - Recurrence: $dp[i][j] = \min_{i<k<j}\{dp[i][k] + dp[k][j]\} + C[i][j]$
   - Condition: $A[i][j-1] \leq A[i][j] \leq A[i+1][j]$
   - Complexity: $\mathcal{O}(n^3) \rightarrow \mathcal{O}(n^2)$

Notes:

- $A[i][j]$ - the smallest k that gives the optimal answer
- $C[i][j]$ - some given cost function

8 Misc

8.1 Mo's Algorithm

Mo's algorithm processes a set of range queries on a static array. Each query is to calculate something base on the array values in a range $[a, b]$. The queries have to be known in advance. Let's divide the array into blocks of size $k = O(\sqrt{n})$. A query $[a_1, b_1]$ is processed before query $[a_2, b_2]$ if $\lfloor \frac{a_1}{k} \rfloor < \lfloor \frac{a_2}{k} \rfloor$ or $\lfloor \frac{a_1}{k} \rfloor = \lfloor \frac{a_2}{k} \rfloor$ and $b_1 < b_2$.

Example problem: counting number of distinct values in a range. We can process the queries in the described order and keep an array count, which knows how many times a certain value has appeared. When moving the boundaries back and forth, we either increase count[x_i] or decrease it. According to value of it, we will know how the number of distinct values has changed (e.g. if count[x_i] has just become 1, then we add 1 to the answer, etc.).

8.2 Ternary Search

```
double ternary_search(double l, double r) {
    while (r - l > eps) {
        double m1 = l + (r - l) / 3;
        double m2 = r - (r - l) / 3;
        double f1 = f(m1);
        double f2 = f(m2);
        if (f1 < f2)
            l = m1;
        else
            r = m2;
    }
    return f(l); //return the maximum of f(x) in [l,
        r]
}
```

8.3 Binary Exponentiation

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

8.4 Builtin GCC Stuff

- __builtin_clz(x): the number of zeros at the beginning of the bit representation.
- __builtin_ctz(x): the number of zeros at the end of the bit representation.
- __builtin_popcount(x): the number of ones in the bit representation.
- __builtin_parity(x): the parity of the number of ones in the bit representation.
- __gcd(x, y): the greatest common divisor of two numbers.
- __int128_t: the 128-bit integer type. Does not support input/output.