ACM-ICPC Team Reference Document

Vilnius University (Šimoliūnaitė, Strakšys, Strimaitis)

Contents

1 Data Structures **1**
1.1 Disjoin Set Union 1
1.2 Fenwick 2D 1
1.3 Fenwick Tree Point Update And Range Query 1
1.4 Fenwick Tree Range Update And Point Query 1
1.5 Fenwick Tree Range Update And Range Query 2
1.6 Implicit Treap 2
1.7 Segment Tree With Lazy Propagation 3
1.8 Segment Tree 4
1.9 Treap 4
1.10 Trie 5

2 General **5**
2.1 Automatic Test 5
2.2 C++ Template 6
2.3 Compilation 6

3 Graphs **6**
3.1 Bipartite Graph 6
3.2 Max Flow With Dinic 8
3.3 Max Flow With Ford Fulkerson 9
3.4 Min Cut 9

4 Math **10**
4.1 Big Integer Multiplication With FFT 10
4.2 Burnside's Lemma 10
4.3 Chinese Remainder Theorem 11
4.4 Euler Totient Function 11
4.5 Extended Euclidean Algorithm 11
4.6 Factorization With Sieve 12
4.7 FFT With Modulo 12
4.8 FFT 13
4.9 Formulas 14
4.10 Linear Sieve 14
4.11 Modular Inverse 14

5 Strings **14**
5.1 Hashing 14

1 Data Structures

1.1 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;
    }
};
```

1.2 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;
    }
};
```

1.3 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);
    }
};
```

1.4 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
    }
};
```

1.5 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);
    }
};
```

1.6 Implicit Treap

```
template <typename T>
struct Node {
    Node* l, *r;
    ll prio, size, sum;
    T val;
    bool rev;
    Node() {}
    Node(T _val) : l(nullptr), r(nullptr), val(_val), size(1), sum(_val), rev(false) {
        prio = rand() ^ (rand() << 15);
    }
};
template <typename T>
struct ImplicitTreap {
    typedef Node<T>* NodePtr;
    int sz(NodePtr n) {
        return n ? n->size : 0;
    }
    ll getSum(NodePtr n) {
        return n ? n->sum : 0;
    }

    void push(NodePtr n) {
        if (n && n->rev) {
            n->rev = false;
            swap(n->l, n->r);
            if (n->l) n->l->rev ^= 1;
            if (n->r) n->r->rev ^= 1;
        }
    }

    void recalc(NodePtr n) {
        if (!n) return;
        n->size = sz(n->l) + 1 + sz(n->r);
        n->sum = getSum(n->l) + n->val + getSum(n->r);
    }

    void split(NodePtr tree, ll key, NodePtr& l, NodePtr& r) {
        push(tree);
        if (!tree) {
            l = r = nullptr;
        }
        else if (key <= sz(tree->l)) {
            split(tree->l, key, l, tree->l);
            r = tree;
        }
        else {
            split(tree->r, key-sz(tree->l)-1, tree->r, r);
            l = tree;
        }
        recalc(tree);
    }

    void merge(NodePtr& tree, NodePtr l, NodePtr r) {
        push(l); push(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, T val, int pos) {
        if (!tree) {
            tree = new Node<T>(val);
            return;
        }
        NodePtr L, R;
        split(tree, pos, L, R);
        merge(L, L, new Node<T>(val));
        merge(tree, L, R);
        recalc(tree);
    }

    void reverse(NodePtr tree, int l, int r) {
        NodePtr t1, t2, t3;
        split(tree, l, t1, t2);
        split(t2, r - l + 1, t2, t3);
        if(t2) t2->rev = true;
        merge(t2, t1, t2);
        merge(tree, t2, t3);
    }

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

    NodePtr fromArray(vector<T> v) {
        NodePtr t = nullptr;
        FOR(i, 0, (int)v.size()) {
            insert(t, v[i], i);
        }
        return t;
    }

    ll calcSum(NodePtr t, int l, int r) {
        NodePtr L, R;
        split(t, l, L, R);
        NodePtr good;
        split(R, r - l + 1, good, L);
        return getSum(good);
    }
};
/* Usage: ImplicitTreap<int> t;
Node<int> tree = t.fromArray(someVector); t.reverse(tree, l, r); ...
*/
```

1.7 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 = _n; 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)
            return -OO;
        if (tl == tr)
            return t[v];
        push(v);
        int tm = (tl + tr) / 2;
        return max(query(v*2, tl, tm, l, min(r, tm)),
```

```
            query(v*2+1, tm+1, tr, max(l, tm+1), r));
    }
};
```

1.8 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]);
        }
    }
};
```

1.9 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;
    }
}
```

1.10 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;
    }
};
```

2 General

2.1 Automatic Test

```
# Linux Bash
# gen, main and stupid have to be compiled beforehand
for((i=1;;++i)); do
    echo $i;
    ./gen $i > genIn;
    diff <(./main < genIn) <(./stupid < genIn) || break;
done

# Windows CMD
@echo off
FOR /L %%I IN (1,1,2147483647) DO (
    echo %%I
    gen.exe %%I > genIn
    main.exe < genIn > mainOut
    stupid.exe < genIn > stupidOut
    FC mainOut stupidOut || goto :eof
)
```

2.2 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;
}
```

2.3 Compilation

```
# Simple compile
g++ -DLOCAL -O2 -o main.exe -std=c++17 -Wall -Wno-unused-result -Wshadow main.cpp
# Debug
g++ -DLOCAL -std=c++17 -Wshadow -Wall -o main.exe main.cpp -fsanitize=address -fsanitize=undefined -fuse-ld=gold -D_GLIBCXX_DEBUG -g
```

3 Graphs

3.1 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 };
    }

};
```

3.2 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);
    }
}
```

3.3 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;
}
```

3.4 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 << " ";
```

4 Math

4.1 Big Integer Multiplication With FFT

```
complex<ld> a[MAX_N], b[MAX_N];
complex<ld> fa[MAX_N], fb[MAX_N], fc[MAX_N];
complex<ld> cc[MAX_N];

string mul(string as, string bs) {
    int sgn1 = 1;
    int sgn2 = 1;
    if (as[0] == '-') {
        sgn1 = -1;
        as = as.substr(1);
    }
    if (bs[0] == '-') {
        sgn2 = -1;
        bs = bs.substr(1);
    }
    int n = as.length() + bs.length() + 1;
    FFT::init(n);
    FOR(i, 0, FFT::pwrN) {
        a[i] = b[i] = fa[i] = fb[i] = fc[i] = cc[i] = 0;
    }
    FOR(i, 0, as.size()) {
        a[i] = as[as.size() - 1 - i] - '0';
    }
    FOR(i, 0, bs.size()) {
        b[i] = bs[bs.size() - 1 - i] - '0';
    }
    FFT::fft(a, fa);
    FFT::fft(b, fb);
    FOR(i, 0, FFT::pwrN) {
        fc[i] = fa[i] * fb[i];
    }
    // turn [0,1,2,...,n-1] into [0, n-1, n-2, ..., 1]
    FOR(i, 1, FFT::pwrN) {
        if (i < FFT::pwrN - i) {
            swap(fc[i], fc[FFT::pwrN - i]);
        }
    }
    FFT::fft(fc, cc);
    ll carry = 0;
    vector<int> v;
    FOR(i, 0, FFT::pwrN) {
        int num = round(cc[i].real() / FFT::pwrN) + carry;
        v.pb(num % 10);
        carry = num / 10;
    }
    while (carry > 0) {
        v.pb(carry % 10);
        carry /= 10;
    }
    reverse(v.begin(), v.end());
    bool start = false;
    ostringstream ss;
    bool allZero = true;
    for (auto x : v) {
        if (x != 0) {
            allZero = false;
            break;
        }
    }
    if (sgn1*sgn2 < 0 && !allZero) ss << "-";
    for (auto x : v) {
        if (x == 0 && !start) continue;
        start = true;
        ss << abs(x);
    }
    if (!start) ss << 0;
    return ss.str();
}
```

4.2 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 $\frac{1}{n}\sum_{k=0}^{n-1} 2^{gcd(k,n)}$.

4.3 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 $0 \le A \le M$.

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 $x \equiv gy + a_1 \pmod{\frac{m_1 m_2}{g}}$.

4.4 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;
}
```

4.5 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:

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

4.6 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;
}
```

4.7 FFT With Modulo

```
bool isGenerator(ll g) {
    if (pwr(g, M - 1) != 1) return false;
    for (ll i = 2; i*i <= M - 1; i++) {
        if ((M - 1) % i == 0) {
            ll q = i;
            if (isPrime(q)) {
                ll p = (M - 1) / q;
                ll pp = pwr(g, p);
                if (pp == 1) return false;
            }
            q = (M - 1) / i;
            if (isPrime(q)) {
                ll p = (M - 1) / q;
                ll pp = pwr(g, p);
                if (pp == 1) return false;
            }
        }
    }
    return true;
}

namespace FFT {
    ll n;
    vector<ll> r;
    vector<ll> omega;
    ll logN, pwrN;

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

    void initOmega() {
        ll g = 2;
        while (!isGenerator(g)) g++;
        ll G = 1;
        g = pwr(g, (M - 1) / pwrN);
        FOR(i, 0, pwrN) {
            omega[i] = G;
            G *= g;
            G %= M;
        }
    }

    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(ll n) {
        FFT::n = n;
        initLogN();
        initArrays();
        initOmega();
        initR();
    }

    void fft(ll a[], ll f[]) {
        for (ll i = 0; i < pwrN; i++) {
            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] % M;
                    f[i + j + k] = f[i + j] - z;
                    f[i + j] += z;
                    f[i + j + k] %= M;
                    if (f[i + j + k] < 0) f[i + j + k] += M;
                    f[i + j] %= M;
                }
            }
        }
    }
}
```

4.8 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;
                }
            }
        }
    }
}
```

4.9 Formulas

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

4.10 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;
        }
    }
}
```

4.11 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 Strings

5.1 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};
    }
};
```