

# ACM-ICPC Team Reference Document

## Tula State University (Basalova, Perezyabov, Provotorin)

### Contents

<b>1</b>	<b>Templates</b>	<b>1</b>
1.1	C++ Template . . . . .	1
1.2	C++ Include . . . . .	2
1.3	Py Template . . . . .	2
<b>2</b>	<b>Data Structures</b>	<b>2</b>
2.1	Disjoint Set Union . . . . .	2
2.2	Segment Tree . . . . .	2
2.3	Segment Tree Propagate . . . . .	3
2.4	Treap . . . . .	3
2.5	Treap K . . . . .	4
2.6	Treap Universal . . . . .	4
2.7	Fenwick Tree . . . . .	5
<b>3</b>	<b>Algebra</b>	<b>5</b>
3.1	Primes Sieve . . . . .	5
3.2	Factorization . . . . .	5
3.3	Euler Totient Function . . . . .	6
3.4	Greatest Common Divisor . . . . .	6
3.5	Binary Operations . . . . .	7
3.6	Matrices . . . . .	7
3.7	Fibonacci . . . . .	7
3.8	Baby Step Giant Step . . . . .	7
3.9	Combinations . . . . .	7
3.10	Permutation . . . . .	8
3.11	Fast Fourier Transform . . . . .	8
3.12	Primitive Roots . . . . .	9
3.13	Number Decomposition . . . . .	9
3.14	Formulae . . . . .	9
<b>4</b>	<b>Geometry</b>	<b>10</b>
4.1	Vector . . . . .	10
4.2	Planimetry . . . . .	10
4.3	Graham . . . . .	10
4.4	Formulae . . . . .	10
<b>5</b>	<b>Stringology</b>	<b>11</b>
5.1	Z Function . . . . .	11
5.2	Manacher . . . . .	11
5.3	Trie . . . . .	11
5.4	Prefix Function . . . . .	11
5.5	Suffix Array . . . . .	12
<b>6</b>	<b>Dynamic Programming</b>	<b>12</b>
6.1	Longest Increasing Subsequence	12

<b>7</b>	<b>Graphs</b>	<b>12</b>
7.1	Graph Implementation . . . . .	12
7.2	Graph Traversing . . . . .	13
7.3	Topological Sort . . . . .	13
7.4	Connected Components . . . . .	13
7.5	2 Sat . . . . .	14
7.6	Bridges . . . . .	14
7.7	Articulation Points . . . . .	15
7.8	Kuhn . . . . .	15
7.9	Kruskal . . . . .	15
7.10	LCA (binary Lifting) . . . . .	16
7.11	LCA RMQ (segtree) . . . . .	16
7.12	Maximum Flow . . . . .	17
7.13	Eulerian Path . . . . .	18
7.14	Eulerian Path Oriented . . . . .	19
7.15	Shortest Paths . . . . .	19
<b>8</b>	<b>Miscellaneous</b>	<b>20</b>
8.1	Ternary Search . . . . .	20

## 1 Templates

### 1.1 C++ Template

```
#include <bits/stdc++.h>

using namespace std;

// DEFINES
#define precision(x) cout << fixed << setprecision(x);
#define fast cin.tie(0); ios::sync_with_stdio(0)
#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()
#define ff first
#define ss second
// #define nl endl
#define nl "\n"
#define sp " "
#define yes "Yes"
#define no "No"
#define int long long

// CONSTANTS
const int INF = 1e18;
// const int MOD = 1e9 + 7;
// const int MOD = 998244353;

// FSTREAMS
ifstream in("input.txt");
ofstream out("output.txt");

// RANDOM
const int RMIN = 1, RMAX = 1e9;

random_device rdev;
```

```
mt19937_64 reng(rdev());
uniform_int_distribution<mt19937_64::result_type> dist(RMIN
, RMAX);
```

```
// CUSTOM HASH
struct custom_hash {
    static uint64_t splitmix64(uint64_t x) {
        // http://xorshift.di.unimi.it/splitmix64.c
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }

    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM = chrono::
            steady_clock::now().time_since_epoch().count();
        return splitmix64(x + FIXED_RANDOM);
    }
};
// USAGE EXAMPLES:
//
// unordered_set<long long, custom_hash> safe_set;
// unordered_map<long long, int, custom_hash> safe_map;
// gp_hash_table<long long, int, custom_hash>
//     safe_hash_table;
//
// for pairs might be used like `3 * a + b` or `a ^ (b >>
//     1)`
```

```
// GLOBALS
```

```
// SOLUTION
void solve() {

}
```

```
// PREPROCESSING
void prepr() { }
```

```
// ENTRANCE
signed main() {
    precision(15);
    fast;
    prepr();
    int t = 1;
    cin >> t;
    while (t--) solve();
}
```

## 1.2 C++ Include

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <random>
#include <cmath>
#include <algorithm>
#include <string>
#include <vector>
#include <set>
#include <unordered_set>
#include <map>
#include <unordered_map>
#include <queue>
#include <deque>
#include <stack>
#include <list>
#include <bitset>
```

## 1.3 Py Template

```
from math import sqrt, ceil, floor, gcd
from random import randint
import sys
```

```
def inpt():
    return sys.stdin.readline().strip()
```

```
input = inpt
```

```
INF = int(1e18)
# MOD = int(1e9 + 7)
# MOD = 998244353
```

```
def solve():
    pass
```

```
t = 1
t = int(input())
for _ in range(t):
    solve()
```

## 2 Data Structures

### 2.1 Disjoint Set Union

```
// Theme: Disjoint Set Union
```

```
struct dsu {
    vector<int> p, size;

    dsu(int n) {
        p.assign(n, 0); size.assign(n, 0);
        for (int i = 0; i < n; i++) {
            p[i] = i;
            size[i] = 1;
        }
    }

    int get(int v) {
        if (p[v] != v) p[v] = get(p[v]);
        return p[v];
    }

    void unite(int u, int v) {
        auto x = get(u), y = get(v);
        if (x == y) return;
        if (size[x] > size[y]) swap(x, y);
        p[x] = y; size[y] += size[x];
    }
};
```

### 2.2 Segment Tree

```
// Theme: Segment Tree
```

```
struct segtree {
    int size;
    vector<int> tree;

    void init(int n) {
        size = 1;
        while (size < n) size <= 1;
        tree.assign(2 * size - 1, 0);
    }

    void build(vector<int> &a, int x, int lx, int rx) {
        if (rx - lx == 1) {
            if (lx < a.size()) tree[x] = a[lx];
            return;
        }
        int m = (lx + rx) / 2;
        build(a, 2 * x + 1, lx, m);
        build(a, 2 * x + 2, m, rx);
        tree[x] = tree[2 * x + 1] + tree[2 * x + 2];
    }
};
```

```

void build(vector<int> &a) {
    init(a.size());
    build(a, 0, 0, size);
}

// Complexity: O(log(n))
void set(int i, int v, int x, int lx, int rx) {
    if (rx - lx == 1) {
        tree[x] = v;
        return;
    }
    int m = (lx + rx) / 2;
    if (i < m) set(i, v, 2 * x + 1, lx, m);
    else set(i, v, 2 * x + 2, m, rx);
    tree[x] = tree[2 * x + 1] + tree[2 * x + 2];
}
void set(int i, int v) {
    set(i, v, 0, 0, size);
}

// Complexity: O(log(n))
int sum(int l, int r, int x, int lx, int rx) {
    if (l <= lx && rx <= r) return tree[x];
    if (l >= rx || r <= lx) return 0;
    int m = (lx + rx) / 2;
    return sum(l, r, 2 * x + 1, lx, m) +
           sum(l, r, 2 * x + 2, m, rx);
}
int sum(int l, int r) {
    return sum(l, r, 0, 0, size);
}
};

```

## 2.3 Segment Tree Propagate

// Theme: Segment Tree With Propagation

```

struct segtree_prop {
    int size;
    vector<int> tree;

    void init(int n) {
        size = 1;
        while (size < n) size <= 1;
        tree.assign(2 * size - 1, 0);
    }

    void build(vector<int> &a, int x, int lx, int rx) {
        if (rx - lx == 1) {
            if (lx < a.size()) tree[x] = a[lx];
            return;
        }
        int m = (lx + rx) / 2;
        build(a, 2 * x + 1, lx, m);
        build(a, 2 * x + 2, m, rx);
        tree[x] = tree[2 * x + 1] + tree[2 * x + 2];
    }

    void build(vector<int> &a) {
        init(a.size());
        build(a, 0, 0, size);
    }

    void push(int x, int lx, int rx) {
        if (rx - lx == 1) return;
        tree[2 * x + 1] += tree[x];
        tree[2 * x + 2] += tree[x];
        tree[x] = 0;
    }

    // Complexity: O(log(n))
    void add(int v, int l, int r, int x, int lx, int rx) {
        push(x, lx, rx);
        if (rx <= l || r <= lx) return;
        if (l <= lx && rx <= r) {
            tree[x] += v;
            return;
        }
        int m = (lx + rx) / 2;
        add(v, l, r, 2 * x + 1, lx, m);
        add(v, l, r, 2 * x + 2, m, rx);
    }

    void add(int v, int l, int r) {
        add(v, l, r, 0, 0, size);
    }
}

```

```

// Complexity: O(log(n))
int get(int i, int x, int lx, int rx) {
    push(x, lx, rx);
    if (rx - lx == 1) return tree[x];
    int m = (lx + rx) / 2;
    if (i < m) return get(i, 2 * x + 1, lx, m);
    else return get(i, 2 * x + 2, m, rx);
}

int get(int i) {
    return get(i, 0, 0, size);
}
};

```

## 2.4 Treap

// Theme: Treap (Tree + Heap)

```

// Node
struct node {
    int key, priority;
    shared_ptr<node> left, right;

    node(int key, int priority = INF) :
        key(key),
        priority(priority == INF ?
            reng() : priority) {}
};

// Treap
struct treap {
    shared_ptr<node> root;

    treap() {}

    treap(int root_key, int root_priority = INF) {
        root = shared_ptr<node>(new node(root_key,
            root_priority));
    }

    treap(shared_ptr<node> rt) {
        root = shared_ptr<node>(rt);
    }

    treap(const treap &tr) {
        root = shared_ptr<node>(tr.root);
    }

    // Complexity: O(log(N))
    pair<treap, treap> split(int k) {
        auto res = split(root, k);
        return { treap(res.ff), treap(res.ss) };
    }

    pair<shared_ptr<node>, shared_ptr<node>> split(
        shared_ptr<node> rt, int k) {
        if (!rt) return { nullptr, nullptr };
        else if (rt->key < k) {
            auto [rt1, rt2] = split(rt->right, k);
            rt->right = rt1;
            return { rt, rt2 };
        }
        else {
            auto [rt1, rt2] = split(rt->left, k);
            rt->left = rt2;
            return { rt1, rt };
        }
    }

    // Complexity: O(log(N))
    treap merge(const treap &tr) {
        root = shared_ptr<node>(merge(root, tr.root));
        return *this;
    }

    shared_ptr<node> merge(shared_ptr<node> rt1, shared_ptr<
        node> rt2) {
        if (!rt1) return rt2;
        if (!rt2) return rt1;
        if (rt1->priority < rt2->priority) {
            rt1->right = merge(rt1->right, rt2);
            return rt1;
        }
        else {
            rt2->left = merge(rt1, rt2->left);
            return rt2;
        }
    }
}

```

```

    }
}
};

```

## 2.5 Treap K

```

// Theme: Treap With Segments

// Node
struct node_k {
    int key, priority, size;
    shared_ptr<node_k> left, right;

    node_k(int key, int priority = INF) :
        key(key),
        priority(priority == INF ?
            reng() : priority),
        size(1) { }

    friend int sz(shared_ptr<node_k> nd) {
        return (nd ? nd->size : 0);
    }

    void upd() {
        size = sz(left) + sz(right) + 1;
    }
};

// Treap
struct treap_k {
    shared_ptr<node_k> root;

    treap_k() { }

    treap_k(int root_key, int root_priority = INF) {
        root = shared_ptr<node_k>(new node_k(root_key,
            root_priority));
    }

    treap_k(shared_ptr<node_k> rt) {
        root = shared_ptr<node_k>(rt);
    }

    treap_k(const treap_k &tr) {
        root = shared_ptr<node_k>(tr.root);
    }

    // Complexity: O(log(N))
    pair<treap_k, treap_k> split_k(int k) {
        auto res = split_k(root, k);
        return { treap_k(res.ff), treap_k(res.ss) };
    }

    pair<shared_ptr<node_k>, shared_ptr<node_k>> split_k(
        shared_ptr<node_k> rt, int k) {
        if (!rt) return { nullptr, nullptr };
        else if (sz(rt) <= k) return { rt, nullptr };
        else if (sz(rt->left) + 1 <= k) {
            auto [rt1, rt2] = split_k(rt->right, k - sz(rt->
                left) - 1);
            rt->right = rt1;
            rt->upd();
            return { rt, rt2 };
        }
        else {
            auto [rt1, rt2] = split_k(rt->left, k);
            rt->left = rt2;
            rt->upd();
            return { rt1, rt };
        }
    }

    // Complexity: O(log(N))
    treap_k merge_k(const treap_k &tr) {
        root = shared_ptr<node_k>(merge_k(root, tr.root));
        return *this;
    }

    shared_ptr<node_k> merge_k(shared_ptr<node_k> rt1,
        shared_ptr<node_k> rt2) {
        if (!rt1) return rt2;
        if (!rt2) return rt1;
        if (rt1->priority < rt2->priority) {
            rt1->right = merge_k(rt1->right, rt2);
            rt1->upd();

```

```

            return rt1;
        }
        else {
            rt2->left = merge_k(rt1, rt2->left);
            rt2->upd();
            return rt2;
        }
    }
};

```

## 2.6 Treap Universal

```

// Theme: Treap (Tree + Heap)
// Supports both explicit and implicit keys (not
// simultaneously ofc)
// Core operations are all O(log n) average

mt19937 rng(378);

struct Node {
    int x, y, size; // "x" is key or payload, "y" is
        priority
    Node* left, * right;

    Node(int val): x(val), y(rng() % 1'000'000'000), size(1)
        , left(nullptr), right(nullptr) {}
};

int get_size(Node* root) {
    if (root == nullptr) return 0;
    return root->size;
}

void update(Node* root) {
    if (root == nullptr) return;
    root->size = get_size(root->left) + 1 + get_size(root->
        right);
}

// split by value (for explicit keys)
pair<Node*, Node*> split(Node* root, int v) {
    if (root == nullptr) return {nullptr, nullptr};
    if (root->x <= v) {
        auto res = split(root->right, v);
        root->right = res.first;
        update(root);
        return {root, res.second};
    }
    else {
        auto res = split(root->left, v);
        root->left = res.second;
        update(root);
        return {res.first, root};
    }
}

// split by size (for implicit keys)
pair<Node*, Node*> split_k(Node* root, int k) {
    if (root == nullptr) return {nullptr, nullptr};
    if (get_size(root) <= k) return {root, nullptr};
    if (k == 0) return {nullptr, root};

    int left_size = get_size(root->left);
    if (left_size >= k) {
        auto res = split_k(root->left, k);
        root->left = res.second;
        update(root);
        return {res.first, root};
    }
    else {
        auto res = split_k(root->right, k - left_size - 1);
        root->right = res.first;
        update(root);
        return {root, res.second};
    }
}

// merge for both explicit and implicit keys
Node* merge(Node* root1, Node* root2) {
    if (root1 == nullptr) return root2;
    if (root2 == nullptr) return root1;

    if (root1->y < root2->y) {
        root1->right = merge(root1->right, root2);
        update(root1);
        return root1;
    }
    else {
        root2->left = merge(root1, root2->left);

```

```

        update(root2);
        return root2;
    }
}

// insert for explicit keys (use split_k for implicit keys)
Node* insert(Node* root, int v) {
    auto subs = split(root, v);
    return merge(merge(subs.first, new Node(v)), subs.second);
}

// debug helper
void print_node(Node* root, bool end = false) {
    if (root->left != nullptr) print_node(root->left);
    cout << root->x << " ";
    if (root->right != nullptr) print_node(root->right);
    if (end) cout << "\n";
}

```

## 2.7 Fenwick Tree

```

// Theme: Fenwick Tree
// Core operations are O(log n)

struct Fenwick {
    vector<int> data;

    explicit Fenwick(int n) {
        data.assign(n + 1, 0);
    }
    explicit Fenwick(vector<int>& arr): Fenwick(arr.size()) {
        for (int i = 1; i <= arr.size(); ++i) {
            add(i, arr[i - 1]);
        }
    }

    // Nested loops (also vector) for multi-dimensional.
    // Also in add().
    // (x & -x) = last non-zero bit
    int sum(int right) {
        int res = 0;
        for (int i = right; i > 0; i -= (i & -i)) {
            res += data[i];
        }
        return res;
    }
    int sum(int left, int right) {
        return sum(right) - sum(left - 1); // inclusion-exclusion principle
    }

    void add(int idx, int x) {
        for (int i = idx; i < data.size(); i += (i & -i)) {
            data[i] += x;
        }
    }

    // CONCEPT (didn't test it). Should work if all real
    // values are non-negative.
    int lower_bound(int s) {
        int k = 0;
        int logn = (int)(log2(data.size() - 1) + 1); //
        maybe rewrite this line
        for (int b = logn; b >= 0; --b) {
            if (k + (1 << b) < data.size() && data[k + (1 <<
                b)] < s) {
                k += (1 << b);
                s -= data[k];
            }
        }
        return k;
    }
};

```

## 3 Algebra

### 3.1 Primes Sieve

```

// Theme: Prime Numbers

// Algorithm: Eratosthenes Sieve
// Complexity: O(N*log(log(N)))

// = 0 - Prime,
// != 0 - Lowest Prime Divisor
auto get_sieve(int n) {
    vector<int> sieve(n); // Sieve
    sieve[0] = sieve[1] = 1;

    for (int i = 2; i * i < n; i++)
        if (!sieve[i])
            for (int j = i * i; j < n; j += i)
                sieve[j] = i;

    return sieve;
}

// Algorithm: Prime Numbers With Sieve
// Complexity: O(N*log(log(N)))

auto get_primes(int n) {
    vector<int> primes, sieve = get_sieve(n);

    for (int i = 2; i < sieve.size(); i++)
        if (!sieve[i])
            primes.push_back(i);

    return primes;
}

// Algorithm: Linear Algorithm
// Complexity: O(N)

// lp[i] = Lowest Prime Divisor
auto get_sieve_primes(int n, vector<int> &primes) {
    vector<int> lp(n);
    lp[0] = lp[1] = 1;

    for (int i = 2; i < n; i++) {
        if (!lp[i]) {
            lp[i] = i;
            primes.push_back(i);
        }
        for (int j = 0; j < primes.size() &&
            primes[j] <= lp[i] &&
            i * primes[j] < n; j++)
            lp[i * primes[j]] = primes[j];
    }

    return lp;
}

```

## 3.2 Factorization

```

// Theme: Factorization

// Algorithm: Trivial Algorithm
// Complexity: O(sqrt(N))

auto factors(int n) {
    vector<int> factors;

    for (int i = 2; i * i <= n; i++) {
        if (n % i)
            continue;
        while (n % i == 0)
            n /= i;
        factors.push_back(i);
    }

    if (n != 1)
        factors.push_back(n);

    return factors;
}

// Algorithm: Factorization With Sieve
// Complexity: O(N*log(log(N)))

auto factors_sieve(int n) {
    vector<int> factors,
    sieve = get_sieve(n + 1);

```

```

while (sieve[n]) {
    factors.push_back(sieve[n]);
    n /= sieve[n];
}

if (n != 1)
    factors.push_back(n);

return factors;
}

// Algorithm: Factorization With Primes
// Complexity: O(sqrt(N)/log(sqrt(N)))

auto factors_primes(int n) {
    vector<int> factors,
        primes = get_primes(n + 1);

    for (auto &i : primes) {
        if (i * i > n)
            break;
        if (n % i)
            continue;
        while (n % i == 0)
            n /= i;
        factors.push_back(i);
    }

    if (n != 1)
        factors.push_back(n);

    return factors;
}

// Algorithm: Ferma Test
// Complexity: O(K*log(N))

bool ferma(int n) {
    if (n == 2)
        return true;

    uniform_int_distribution<int> distA(2, n - 1);

    for (int i = 0; i < 1000; i++) {
        int a = distA(reng);
        if (gcd(a, n) != 1 ||
            binpow(a, n - 1, n) != 1)
            return false;
    }

    return true;
}

// Algorithm: Pollard Rho Algorithm
// Complexity: O(N^(1/4))

int get_random_number(int l, int r) {
    random_device random_device;
    mt19937 generator(random_device());
    uniform_int_distribution<int> distribution(l, r);

    return distribution(generator);
}

int f(int x, int c, int n) {
    return ((x * x + c) % n);
}

int ff(int n) {
    int g = 1;
    for (int i = 0; i < 5; i++) {
        int x = get_random_number(1, n);
        int c = get_random_number(1, n);
        int h = 0;
        while (g == 1) {
            x = f(x, c, n) % n;
            int y = f(f(x, c, n), c, n) % n;
            g = gcd(abs(x - y), n);
            if (g == n) {
                g = 1;
            }
            h++;
        }

        if (h > 4 * (int)pow(n, 1.0 / 4)) {
            break;
        }
    }
}

```

```

    if (g > 1) {
        return g;
    }
}

return -1;
}

signed main() {
    // ...read n...
    vector<int> a;
    while (n > 1) {
        int m = ff(n);
        if (m > 0) {
            n = n / m;
            a.push_back(m);
        } else {
            break;
        }
    }

    vector<int> ans;
    a.push_back(n);

    for (auto &it : a) {
        int i = 2;
        int m = it;
        while (i * i <= m) {
            if (m % i == 0) {
                ans.push_back(i);
                m = m / i;
            } else {
                i += 1;
            }
        }

        ans.push_back(m);
    }

    sort(all(ans));
}

```

### 3.3 Euler Totient Function

```

// Theme: Euler Totient Function

// Algorithm: Euler Product Formula
// Complexity: O(sqrt(N))

// phi = n(1 - 1 / pi), i = 1, ...
int phi(int n) {
    if (n == 1) return 1;

    auto f = factors(n);

    int res = n;
    for (auto &p : f)
        res -= res / p;

    return res;
}

```

### 3.4 Greatest Common Divisor

```

// Theme: Greatest Common Divisor

// Algorithm: Simple Euclidean Algorithm
// Complexity: O(log(N))

int gcd(int a, int b) {
    while (b) {
        a %= b;
        swap(a, b);
    }
    return a;
}

// Algorithm: Extended Euclidean Algorithm
// Complexity: O(log(N))

// d = gcd(a, b)
// x * a + y * b = d

```

```
// returns {d, x, y}
vector<int> euclid(int a, int b) {
    if (!a) return { b, 0, 1 };
    auto v = euclid(b % a, a);
    int d = v[0], x = v[1], y = v[2];
    return { d, y - (b / a) * x, x };
}
```

## 3.5 Binary Operations

// Theme: Binary Operations

// Algorithm: Binary Multiplication  
// Complexity:  $O(\log(b))$

```
int binmul(int a, int b, int p = 0) {
    int res = 0;
    while (b) {
        if (b & 1) res = p ? (res + a) % p : (res + a);
        a = p ? (a + a) % p : (a + a);
        b >>= 1;
    }
    return res;
}
```

// Algorithm: Binary Exponentiation  
// Complexity:  $O(\log(b))$

```
int binpow(int a, int b, int p = 0) {
    int res = 1;
    while (b) {
        if (b & 1) res = p ? (res * a) % p : (res * a);
        a = p ? (a * a) % p : (a * a);
        b >>= 1;
    }
    return res;
}
```

## 3.6 Matrices

// Theme: Matrix Operations

```
template <typename T>
using row = vector<T>;
template <typename T>
using matrix = vector<vector<T>>;
```

// Algorithm: Matrix-Matrix Multiplication  
// Complexity:  $O(N * K * M)$

```
auto m_prod(matrix<int> &a, matrix<int> &b, int p = 0) {
    int n = a.size(), k = a[0].size(), m = b[0].size();

    matrix<int> res(n, row<int>(m));

    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            for (int z = 0; z < k; z++)
                res[i][j] = p ? (res[i][j] + a[i][z] * b[z][j])
                % p : (res[i][j] + a[i][z] * b[z][j]);

    return res;
}
```

// Algorithm: Matrix-Vector Multiplication  
// Complexity:  $O(N * M)$

```
auto m_prod(matrix<int> &a, row<int> &b, int p = 0) {
    int n = a.size(), m = b.size();

    row<int> res(n);

    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            res[i] = p ? (res[i] + a[i][j] * b[j] % p) % p
            : (res[i] + a[i][j] * b[j]);

    return res;
}
```

// Algorithm: Fast Matrix Exponentiation

// Complexity:  $O(N^3 * \log(K))$

```
auto m_binpow(matrix<int> a, int x, int p = 0) {
    int n = a.size();

    matrix<int> res(n, row<int>(n));
    for (int i = 0; i < n; i++) res[i][i] = 1;

    while (x) {
        if (x & 1) res = m_prod(res, a, p);
        a = m_prod(a, a, p);
        x >>= 1;
    }

    return res;
}
```

## 3.7 Fibonacci

// Theme: Fibonacci Sequence

// Algorithm: Fibonacci Numbers With Matrix Exponentiation  
// Complexity:  $O(\log(N))$

```
int fibonacci(int n) {
    row<int> first_two = { 1, 0 };
    if (n <= 2) return first_two[2 - n];

    matrix<int> fib(2, row<int>(2, 0));
    fib[0][0] = 1; fib[0][1] = 1;
    fib[1][0] = 1; fib[1][1] = 0;

    fib = m_binpow(fib, n - 2);

    row<int> last_two = m_prod(fib, first_two);

    return last_two[0];
}
```

## 3.8 Baby Step Giant Step

// Theme: Discrete Logarithm

// Algorithm: Baby-Step Giant-Step Algorithm  
// Complexity:  $O(\sqrt{p} * \log(p))$

```
//  $a^x \equiv b \pmod{p}$ ,  $(a, p) = 1$ 
//  $a^{(i * m + j)} \equiv b \pmod{p}$ ,  $m = \text{ceil}(\sqrt{p})$ 
//  $a^{(i * m)} \equiv b * a^{(-j)} \pmod{p}$ 
int baby_giant_step(int a, int b, int p) {
    //  $a^{(-1)} \equiv a^{(p-2)} \pmod{p}$ 
    int m = ceil(sqrt(p)), _a = binpow(a, p - 2, p);

    //  $s[b * a^{(-j)}] = j$ 
    unordered_map<int, int> s;
    for (int j = 0, t = b; j < m; j++, t = t * _a % p) s[t] = j;

    for (int i = 0; i < m; i++) {
        //  $s.\text{find}(a^{(i * m)})$ 
        auto f = s.find(binpow(a, i * m, p));
        //  $i * m + j$ 
        if (f != s.end()) return i * m + f->ss;
    }

    return -1;
}
```

## 3.9 Combinations

// Theme: Combination Number

// Algorithm: Online Multiplication-Division  
// Complexity:  $O(k)$

```
//  $C_n^k$  - from n by k
int C(int n, int k) {
    int res = 1;
```

```

    for (int i = 1; i <= k; i++) {
        res *= n - k + i;
        res /= i;
    }

    return res;
}

// Algorithm: Pascal Triangle Preprocessing
// Complexity: O(N^2)

auto pascal(int n) {
    // C[i][j] = C_{i+j}^i
    vector<vector<int>> C(n + 1, vector<int>(n + 1, 1));
    for (int i = 1; i < n + 1; i++)
        for (int j = 1; j < n + 1; j++)
            C[i][j] = C[i - 1][j] + C[i][j - 1];

    return C;
}

```

### 3.10 Permutation

```

// Theme: Permutations

// Algorithm: Next Lexicological Permutation
// Complexity: O(N)

bool perm(vector<int> &v) {
    int n = v.size();

    for (int i = n - 1; i >= 1; i--) {
        if (v[i - 1] < v[i]) {
            reverse(v.begin() + i, v.end());

            int j = distance(v.begin(),
                upper_bound(v.begin() + i, v.end(), v[i - 1]));

            swap(v[i - 1], v[j]);
            return true;
        }
    }

    return false;
}

```

### 3.11 Fast Fourier Transform

```

// Theme: Fast Fourier Transform

// Algorithm: Fast Fourier Transform (Complex)
// Complexity: O(N*log(N))

using cd = complex<double>;
const double PI = acos(-1);

auto fft(vector<cd> a, bool invert = 0) {
    // n = 2 ^ x
    int n = a.size();

    // Bit-Reversal Permutation (0000, 1000, 0100, 1100,
    // 0010, ...)
    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j >= bit; bit >>= 1) j -= bit;
        j += bit;
        if (i < j) swap(a[i], a[j]);
    }

    for (int len = 2; len <= n; len <<= 1) {
        // Complex Root Of One
        double ang = 2 * PI / len * (invert ? -1 : 1);
        cd lroot(cos(ang), sin(ang));

        for (int i = 0; i < n; i += len) {
            cd root(1);
            for (int j = 0; j < len / 2; j++) {
                cd u = a[i + j], v = a[i + j + len / 2] * root;
                a[i + j] = (u + v);
                a[i + j + len / 2] = (u - v);
            }
        }
    }
}

```

```

        root = (root * lroot);
    }
}

if (invert) {
    for (int i = 0; i < n; i++) a[i] /= n;
}

return a;
}

// Module (7340033 = 7 * (2 ^ 20) + 1)
// Primitive Root (5 ^ (2 ^ 20) == 1 mod 7340033)
// Inverse Primitive Root (5 * 4404020 == 1 mod 7340033)
// Maximum Degree Of Two (2 ^ 20)

const int mod = 7340033;
const int proot = 5;
const int proot_1 = 4404020;
const int pw = 1 << 20;

// Algorithm: Discrete Fourier Transform (Inverse Roots)
// Complexity: O(N*log(N))

auto fft(vector<int> a, bool invert = 0) {
    // n = 2 ^ x
    int n = a.size();

    // Bit-Reversal Permutation (0000, 1000, 0100, 1100,
    // 0010, ...)
    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j >= bit; bit >>= 1) j -= bit;
        j += bit;
        if (i < j) swap(a[i], a[j]);
    }

    for (int len = 2; len <= n; len <<= 1) {
        // Primitive Root Or Inverse Root (Inverse
        // Transform)
        int lroot = invert ? proot_1 : proot;

        // Current Primitive Root
        lroot = binpow(lroot, pw / len, mod);

        for (int i = 0; i < n; i += len) {
            int root = 1;
            for (int j = 0; j < len / 2; j++) {
                int u = a[i + j], v = a[i + j + len / 2] *
                    root % mod;
                a[i + j] = (u + v) % mod;
                a[i + j + len / 2] = (u - v + mod) % mod;
                root = (root * lroot) % mod;
            }
        }

        if (invert) {
            int _n = binpow(n, mod - 2, mod);
            for (int i = 0; i < n; i++) a[i] = (a[i] * _n) % mod;
        }

        return a;
    }
}

// Algorithm: Discrete Fourier Transform
// Complexity: O(N*log(N))

auto fft(vector<int> &a, bool invert = 0) {
    // n = 2 ^ x
    int n = a.size();

    // Bit-Reversal Permutation (0000, 1000, 0100, 1100,
    // 0010, ...)
    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j >= bit; bit >>= 1) j -= bit;
        j += bit;
        if (i < j) swap(a[i], a[j]);
    }

    for (int len = 2; len <= n; len <<= 1) {
        // Current Primitive Root
        int lroot = binpow(proot, pw / len, mod);
    }
}

```



```

    for (int i = 0; i < n; i += len) {
        int root = 1;
        for (int j = 0; j < len / 2; j++) {
            int u = a[i + j], v = a[i + j + len / 2] *
                root % mod;
            a[i + j] = (u + v) % mod;
            a[i + j + len / 2] = (u - v + mod) % mod;
            root = (root * lroot) % mod;
        }
    }

    if (invert) {
        reverse(a.begin() + 1, a.end());
        int _n = binpow(n, mod - 2, mod);
        for (int i = 0; i < n; i++) a[i] = (a[i] * _n) % mod;
    }

    return a;
}

```

### 3.12 Primitive Roots

```

// Module (7 == 3 * (2 ^ 1) + 1)
// Primitive Root (3)
// Primitive Root {2 ^ 1} (6)
// Inverse Root {2 ^ 1} (6)
// Degree Of Two (2)

```

```

// const int mod = 7
// const int proot = 6
// const int proot_1 = 6
// const int pw = 1 << 1

```

```

// Module (13 == 3 * (2 ^ 2) + 1)
// Primitive Root (2)
// Primitive Root {2 ^ 2} (8)
// Inverse Root {2 ^ 2} (5)
// Degree Of Two (4)

```

```

// const int mod = 13
// const int proot = 8
// const int proot_1 = 5
// const int pw = 1 << 2

```

### 3.13 Number Decomposition

```

// Theme: Integer Numbers Decomposition With Composite
Module

```

```

// Module
// m = (p1 ^ m1) * (p2 ^ m2) * ... * (pn ^ mn)
int m;
// Prime Divisors Of Module
vector<int> p;

```

```

struct num {
    // GCD(x, m) = 1
    int x;
    // Powers Of Primes
    vector<int> a;

    num() : x(0), a(vector<int>(p.size())) { }

    // n = (p1 ^ a1) * (p2 ^ a2) * ... * (pn ^ an) * x
    num(int n) : x(0), a(vector<int>(p.size())) {
        if (!n) return;
        for (int i = 0; i < p.size(); i++) {
            int ai = 0;
            while (n % p[i] == 0) {
                n /= p[i];
                ai++;
            }
            a[i] = ai;
        }
        x = n;
    }
}

```

```

num operator*(const num &nm) {
    vector<int> new_a(p.size());
    for (int i = 0; i < p.size(); i++)
        new_a[i] = a[i] + nm.a[i];
    num res; res.a = new_a;
    res.x = x * nm.x % m;
    return res;
}

```

```

num operator/(const num &nm) {
    vector<int> new_a(p.size());
    for (int i = 0; i < p.size(); i++)
        new_a[i] = a[i] - nm.a[i];
    num res; res.a = new_a;
    int g = euclid(nm.x, m)[1];
    g += m; g %= m;
    res.x = x * g % m;
    return res;
}

```

```

int toint() {
    int res = x;
    for (int i = 0; i < p.size(); i++)
        res = res * binpow(p[i], a[i], m) % m;
    return res;
}
};

```

### 3.14 Formulae

#### Combinations.

$$C_n^k = \frac{n!}{(n-k)!k!}$$

$$C_n^0 + C_n^1 + \dots + C_n^n = 2^n$$

$$C_{n+1}^{k+1} = C_n^{k+1} + C_n^k$$

$$C_n^k = \frac{n}{k} C_{n-1}^{k-1}$$

#### Strling approximation.

$$n! \approx \sqrt{2\pi n} \frac{n^n}{e^n}$$

#### Euler's theorem.

$$a^{\phi(m)} \equiv 1 \pmod{m}, \gcd(a, m) = 1$$

#### Ferma's little theorem.

$$a^{p-1} \equiv 1 \pmod{p}, \gcd(a, p) = 1, p - \text{prime}.$$

#### Catalan number.

$$C_0 = 0, C_n = \sum_{i=0}^{n-1} C_i C_{n-1-i}$$

$$C_n = \frac{2(2n-1)}{n+1} C_{n-1}$$

$$C_n = \frac{(2n)!}{n!(n+1)!}$$

#### Arithmetic progression.

$$S_n = \frac{a_1 + a_n}{2} n = \frac{2a_1 + d(n-1)}{2} n$$

#### Geometric progression.

$$S_n = \frac{b_1(1-q^n)}{1-q} n$$

#### Infinitely decreasing geometric progression.

$$S_n = \frac{b_1}{1-q} n$$

#### Sums.

$$\sum_{i=1}^n i = \frac{n(n+1)}{2},$$

$$\sum_{i=1}^n i^2 = \frac{n(2n+1)(n+1)}{6},$$

$$\sum_{i=1}^n i^3 = \frac{n^2(n+1)^2}{4},$$

$$\sum_{i=1}^n i^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30},$$

$$\sum_{i=a}^b c^i = \frac{c^{b+1}-c^a}{c-1}, c \neq 1.$$

## 4 Geometry

### 4.1 Vector

// Theme: Mathematical 3-D Vector

```
template <typename T>
struct vec {
    T x, y, z;
    vec(T x = 0, T y = 0, T z = 0) : x(x), y(y), z(z) {}
    vec<T> operator+(const vec<T> &v) const {
        return vec<T>(x + v.x, y + v.y, z + v.z);
    }
    vec<T> operator-(const vec<T> &v) const {
        return vec<T>(x - v.x, y - v.y, z - v.z);
    }
    vec<T> operator*(T k) const {
        return vec<T>(k * x, k * y, k * z);
    }
    friend vec<T> operator*(T k, const vec<T> &v) {
        return vec<T>(v.x * k, v.y * k, v.z * k);
    }
    vec<T> operator/(T k) {
        return vec<T>(x / k, y / k, z / k);
    }
    T operator*(const vec<T> &v) const {
        return x * v.x + y * v.y + z * v.z;
    }
    vec<T> operator^(const vec<T> &v) const {
        return { y * v.z - z * v.y, z * v.x - x * v.z, x * v
            .y - y * v.x };
    }
    auto operator<=>(const vec<T> &v) const = default;
    bool operator==(const vec<T> &v) const = default;
    T norm() const {
        return x * x + y * y + z * z;
    }
    double abs() const {
        return sqrt(norm());
    }
    double cos(const vec<T> &v) const {
        return ((*this) * v) / (abs() * v.abs());
    }
    friend ostream &operator<<(ostream &out, const vec<T> &v
        ) {
        return out << v.x << sp << v.y << sp << v.z;
    }
    friend istream &operator>>(istream &in, vec<T> &v) {
        return in >> v.x >> v.y >> v.z;
    }
};
```

### 4.2 Planimetry

// Theme: Planimetry Objects

```
// Point
template <typename T>
struct point {
    T x, y;

    point() : x(0), y(0) {}
    point(T x, T y) : x(x), y(y) {}
};

// Rectangle
template <typename T>
struct rectangle {
    point<T> ld, ru;

    rectangle(const point<T> &ld, const point<T> &ru) :
        ld(ld), ru(ru) {}
};
```

### 4.3 Graham

// Theme: Convex Hull

// Algorithm: Graham Algorithm  
// Complexity:  $O(N \cdot \log(N))$

```
auto graham(const vector<vec<int>> &points) {
    vec<int> p0 = points[0];

    for (auto p : points)
        if (p.y < p0.y ||
            p.y == p0.y && p.x > p0.x)
            p0 = p;

    for (auto &p : points) {
        p.x -= p0.x;
        p.y -= p0.y;
    }

    sort(all(points), [] (vec<int> &p1, vec<int> &p2) {
        return (p1 ^ p2).z > 0 ||
            (p1 ^ p2).z == 0 && p1.norm() > p2.norm(); });

    vector<vec<int>> hull;
    for (auto &p : points) {
        while (hull.size() >= 2 &&
            (((p - hull.back()) ^ (hull[hull.size() - 1] - hull[
                hull.size() - 2]))).z >= 0)
            hull.pop_back();
        hull.push_back(p);
    }

    for (auto &p : hull) {
        p.x += p0.x;
        p.y += p0.y;
    }

    return hull;
}
```

### 4.4 Formulae

#### Triangles.

Radius of circumscribed circle:

$$R = \frac{abc}{4S}.$$

Radius of inscribed circle:

$$r = \frac{S}{p}.$$

Side via medians:

$$a = \frac{2}{3}\sqrt{2(m_b^2 + m_c^2) - m_a^2}.$$

Median via sides:

$$m_a = \frac{1}{2}\sqrt{2(b^2 + c^2) - a^2}.$$

Bisector via sides:

$$l_a = \frac{2\sqrt{bcp(p-a)}}{b+c}.$$

Bisector via two sides and angle:

$$l_a = \frac{2bc \cos \frac{\alpha}{2}}{b+c}.$$

Bisector via two sides and divided side:

$$l_a = \sqrt{bc - a_b a_c}.$$

#### Right triangles.

$a, b$  - cathets,  $c$  - hypotenuse.

$h$  - height to hypotenuse, divides  $c$  to  $c_a$  and

$$c_b. \quad \begin{cases} h^2 = c_a \cdot c_b, \\ a^2 = c_a \cdot c, \\ b^2 = c_b \cdot c. \end{cases}$$

#### Quadrangles.

Sides of circumscribed quadrangle:

$$a + c = b + d.$$

Square of circumscribed quadrangle:

$$S = \frac{Pr}{2} = pr.$$

Angles of inscribed quadrangle:

$$\alpha + \gamma = \beta + \delta = 180^\circ.$$

Square of inscribed quadrangle:

$$S = \sqrt{(p-a)(p-b)(p-c)(p-d)}.$$

## Circles.

Intersection of circle and line:

$$\begin{cases} (x - x_0)^2 + (y - y_0)^2 = R^2 \\ y = ax + b \end{cases}$$

Task comes to solution of  $\alpha x^2 + \beta x + \gamma = 0$ ,

where

$$\begin{cases} \alpha = (1 + a^2), \\ \beta = (2a(b - y_0) - 2x_0), \\ \gamma = (x_0^2 + (b - y_0)^2 - R^2). \end{cases}$$

Intersection of circle and circle:

$$\begin{cases} (x - x_0)^2 + (y - y_0)^2 = R_0^2 \\ (x - x_1)^2 + (y - y_1)^2 = R_1^2 \end{cases}$$

$$y = \frac{1}{2} \frac{(R_1^2 - R_0^2) + (x_0^2 - x_1^2) + (y_0^2 - y_1^2)}{y_0 - y_1} - \frac{x_0 - x_1}{y_0 - y_1} x$$

Task comes to intersection of circle and line.

## 5 Stringology

### 5.1 Z Function

// Theme: Z-Function

// Algorithm: Linear Algorithm  
// Complexity: O(N)

```
auto z_func(const string &s) {
    int n = s.size();
    vector<int> z(n);

    for (int i = 1, l = 0, r = 0; i < n; i++) {
        if (i <= r) z[i] = min(r - i + 1, z[i - l]);

        while (i + z[i] < n && s[z[i]] == s[i + z[i]]) z[i]++;

        if (i + z[i] - 1 > r) {
            l = i;
            r = i + z[i] - 1;
        }
    }

    return z;
}
```

### 5.2 Manacher

// Theme: Palindromes

// Algorithm: Manacher Algorithm  
// Complexity: O(N)

```
int manacher(const string &s) {
    int n = s.size();
    vector<int> d1(n), d2(n);

    for (int i = 0, l = 0, r = -1; i < n; i++) {
        int k = i > r ? 1 : min(d1[l + r - i], r - i + 1);
        while (i + k < n && i - k >= 0 && s[i + k] == s[i - k]) k++;
        d1[i] = k;
        if (i + k - 1 > r) {

```

```
            l = i - k + 1;
            r = i + k - 1;
        }
    }

    for (int i = 0, l = 0, r = -1; i < n; i++) {
        int k = i > r ? 0 : min(d2[l + r - i + 1], r - i + 1);
        while (i + k < n && i - k - 1 >= 0 && s[i + k] == s[i - k - 1]) k++;
        d2[i] = k;
        if (i + k - 1 > r) {
            l = i - k;
            r = i + k - 1;
        }
    }

    int res = 0;
    for (int i = 0; i < n; i++) {
        res += d1[i] + d2[i];
    }

    return res;
}
```

### 5.3 Trie

// Theme: Trie

// Algorithm: Aho-Corasick  
// Complexity: O(N)

```
struct trie {
    // Vertex
    struct vertex {
        vector<int> next;
        bool leaf;
    };

    // Alphabet size
    static const int N = 26;
    // Maximum Vertex Number
    static const int MX = 2e5 + 1;

    // Vertices Vector
    vector<vertex> t;
    int sz;

    trie(): sz(1) {
        t.resize(MX);
        t[0].next.assign(N, -1);
    }

    void add_str(const string &s) {
        int v = 0;
        for (int i = 0; i < s.length(); i++) {
            char c = s[i] - 'a';
            if (t[v].next[c] == -1) {
                t[sz].next.assign(N, -1);
                t[v].next[c] = sz++;
            }
            v = t[v].next[c];
        }
        t[v].leaf = true;
    }
};
```

### 5.4 Prefix Function

// Theme: Prefix function

// Algorithm: Knuth-Morris-Pratt Algorithm  
// Complexity: O(N)

```
auto pref_func(const string &s) {
    int n = s.size();
    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;
}

```

## 5.5 Suffix Array

```

// Theme: Suffix array

// Algorithm: Binary Algorithm With Count Sort
// Complexity: O(N*log(N))

void count_sort(vector<int> &p, vector<int> &c) {
    int n = p.size();
    vector<int> cnt(n), p_new(n), pos(n);

    for (auto &x : c) cnt[x]++;

    pos[0] = 0;
    for (int i = 1; i < n; i++)
        pos[i] = pos[i - 1] + cnt[i - 1];

    for (auto &x : p) {
        int i = c[x];
        p_new[pos[i]] = x;
        pos[i]++;
    }

    p = p_new;
}

auto suffix_array(const string &str) {
    string s = str + '$';
    int n = s.size();

    vector<int> p(n), c(n);
    vector<pair<char, int>> a(n);

    for (int i = 0; i < n; i++) a[i] = { str[i], i };

    sort(a.begin(), a.end());

    for (int i = 0; i < n; i++) p[i] = a[i].second;

    c[p[0]] = 0;
    for (int i = 1; i < n; i++)
        c[p[i]] = c[p[i - 1]] + (a[i].first != a[i - 1].first);

    int k = 0;
    while ((1 << k) < n) {
        for (int i = 0; i < n; i++)
            p[i] = (p[i] - (1 << k) + n) % n;

        count_sort(p, c);

        vector<int> c_new(n);

        c_new[p[0]] = 0;
        for (int i = 1; i < n; i++) {
            pair<int, int> prev = { c[p[i - 1]], c[(p[i - 1] + (1 << k)) % n] };
            pair<int, int> now = { c[p[i]], c[(p[i] + (1 << k)) % n] };
            c_new[p[i]] = c_new[p[i - 1]] + (now != prev);
        }

        c = c_new;
        k++;
    }

    return p;
}

```

## 6 Dynamic Programming

### 6.1 Longest Increasing Subsequence

```

// Theme: Longest Increasing Subsequence

// Algorithm: LIS via binary search
// Complexity: O(N*log(N))

vector<int> lis(vector<int> &arr) {
    int n = arr.size();
    vector<int> dp(n + 1, INF); dp[0] = -INF;
    vector<int> pos(n + 1, -1), previous(n, -1);
    int length = 0;
    for (int i = 0; i < n; ++i) {
        int j = lower_bound(dp.begin(), dp.end(), arr[i]) - dp.begin();
        if (dp[j - 1] < arr[i] && arr[i] < dp[j]) {
            dp[j] = arr[i];
            pos[j] = i;
            previous[i] = pos[j - 1];
            length = max(length, j);
        }
    }

    vector<int> res;
    int p = pos[length];
    while (p != -1) {
        res.push_back(arr[p]);
        p = previous[p];
    }
    reverse(res.begin(), res.end());
    return res;
}

```

## 7 Graphs

### 7.1 Graph Implementation

```

// Theme: Graph Implementation

////////////////////////////////////
// Adjacency List (Unoriented)
////////////////////////////////////

int sz;

vector<vector<int>> graph;

graph.assign(n, {});

for (int i = 0; i < n; i++) {
    int u, v; cin >> u >> v; --u --v;
    graph[u].push_back(v);
    graph[v].push_back(u);
}

////////////////////////////////////
// Adjacency List (Oriented)
////////////////////////////////////

int sz;

vector<vector<int>> graph;
vector<vector<int>> rgraph;

graph.assign(n, {});
rgraph.assign(n, {});

for (int i = 0; i < n; i++) {
    int u, v; cin >> u >> v; --u --v;
    graph[u].push_back(v);
    rgraph[v].push_back(u);
}

////////////////////////////////////
// Edges List (Unoriented)
////////////////////////////////////

```

```

int sz;

vector<pair<int, int>> edges;
vector<vector<int>> graph;

graph.assign(n, {});

for (int i = 0; i < n; i++) {
    int u, v; cin >> u >> v; --u; --v;
    edges.push_back({ u, v });
    graph[u].push_back(i);
    graph[v].push_back(i);
}

//////////
// Edges List + Structure (Unoriented)
//////////

struct edge {
    int u, v, w;
    edge(int u, int v, int w = 0)
        : u(u), v(v), w(w) { }
};

int sz;

vector<edge> edges;
vector<vector<int>> graph;

graph.assign(n, {});

for (int i = 0; i < n; i++) {
    int u, v, w; cin >> u >> v >> w; --u; --v;
    edges.push_back({ u, v, w });
    graph[u].push_back(i);
    graph[v].push_back(i);
}

//////////
// Edges List + Structure + Net Flows (Oriented)
//////////

struct edge {
    int to, cap, flow, weight;
    edge(int to, int cap, int flow = 0, int weight = 0)
        : to(to), cap(cap), flow(flow), weight(weight) { }
    int res() {
        return cap - flow;
    }
};

int sz;

vector<edge> edges;
vector<vector<int>> fgraph;

fgraph.assign(n, {});

void add_edge(int u, int v, int limit, int flow = 0, int
    weight = 0) {
    fgraph[u].push_back(edges.size());
    edges.push_back({ v, limit, flow, weight });
    fgraph[v].push_back(edges.size());
    edges.push_back({ u, 0, 0, -weight });
}

//////////
// Adjacency Matrix
//////////

vector<vector<int>> graph;

for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        cin >> graph[i][j];

```

## 7.2 Graph Traversing

```

// Theme: Graph Traversing

vector<vector<int>> graph;
vector<int> used;

// Algorithm: Depth-First Search (Adjacency List)
// Complexity: O(N + M)

```

```

void dfs(int cur, int p = -1) {
    used[cur] = 1;

    for (auto &to : graph[cur]) {
        if (to == p || used[to]) continue;
        dfs(to, cur);
    }
}

// Algorithm: Breadth-First Search (Adjacency List)
// Complexity: O(N + M)

void bfs(int u) {
    queue<int> q; q.push(u);

    while (q.size()) {
        int cur = q.front(); q.pop();

        for (auto &to : graph[cur]) {
            if (used[to]) continue;
            q.push(to);
        }
    }
}

```

## 7.3 Topological Sort

```

// Theme: Topological Sort

vector<vector<int>> graph;
vector<int> used;

// Algorithm: Topological Sort
// Complexity: O(N + M)

vector<int> topsort;

void dfs_topsort(int cur, int p = -1) {
    used[cur] = 1;

    for (auto &to : graph[cur]) {
        if (to == p || used[to]) continue;
        dfs(to, cur);
    }

    topsort.push_back(cur);
}

for (int u = 0; u < n; u++)
    if (!used[u])
        dfs_topsort(u);

reverse(all(topsort));

```

## 7.4 Connected Components

```

// Theme: Connectivity Components

vector<vector<int>> graph;
vector<int> used;

// Algorithm: Connected Components
// Complexity: O(N + M)

vector<vector<int>> cc;

void dfs_cc(int cur, int p = -1) {
    used[cur] = 1;
    cc.back().push_back(cur);

    for (auto &to : graph[cur]) {
        if (to == p || used[to]) continue;
        dfs_cc(to, cur);
    }
}

for (int u = 0; u < n; u++)
    if (!used[u])
        dfs_cc(u);

// Algorithm: Strongly Connected Components

```

```
// Complexity: O(N + M)

vector<vector<int>> rgraph;

vector<vector<int>> topsort;

vector<vector<int>> scc;

void dfs_scc(int cur, int p = -1) {
    used[cur] = 1;
    scc.back().push_back(cur);

    for (auto &to : rgraph[cur]) {
        if (to == p || used[to]) continue;
        dfs_scc(to, cur);
    }
}

for (auto &u: topsort)
    if (!used[u])
        dfs_scc(u);
```

## 7.5 2 Sat

```
// Theme: 2-SAT

// Algorithm: Adding Edges To 2-SAT

vector<vector<int>> ts_graph;
vector<vector<int>> ts_rgraph;
vector<int> used;
vector<int> top_sort;

// Vertex By Var Number
int to_vert(int x) {
    if (x < 0) {
        return ((abs(x) - 1) << 1) ^ 1;
    }
    else {
        return (x - 1) << 1;
    }
}

// Adding Implication
void add_impl(int a, int b) {
    ts_graph[a].insert(b);
    ts_rgraph[b].insert(a);
}

// Adding Disjunction
void add_or(int a, int b) {
    add_impl(a ^ 1, b);
    add_impl(b ^ 1, a);
}

// topsort
void dfs(int v) {
    used[v] = 1;
    for (auto to: ts_graph[v]) {
        if (!used[to]) dfs(to);
    }
    top_sort.push_back(v);
}

// scc
vector<vector<long long int>> scc;
void dfs_scc(long long int cur, long long int p = -1) {
    used[cur] = 1;
    scc.back().push_back(cur);
    for (auto to : rgr[cur]) {
        if (to == p || used[to]) continue;
        dfs_scc(to, cur);
    }
}

int main() {
    ...
    used.resize(n, 0);
    for (i = 0; i < n; i++) {
        if (!used[i]) dfs(i);
    }
    reverse(top_sort.begin(), top_sort.end());
    for (auto it: top_sort) {
        if (!used[u]) {
            scc.push_back({});
            dfs_scc(u);
        }
    }
}
```

```
    }
}
vector<long long int> v_scc;
v_scc.assign(2 * n, -1);

for (int i = 0; i < scc.size(); i++)
    for (auto &u : scc[i])
        v_scc[u] = i;

vector<long long int> values(2 * n, -1);

for (int i = 0; i < 2 * n; i += 2)
    if (v_scc[i] == v_scc[i ^ 1]) {
        cout << "NO\n";
        return 0;
    }
    else {
        if (v_scc[i] < v_scc[i ^ 1]) {
            values[i] = 0;
            values[i ^ 1] = 1;
        }
        else {
            values[i] = 1;
            values[i ^ 1] = 0;
        }
    }
}
}
```

## 7.6 Bridges

```
// Theme: Bridges And ECC

vector<pair<int, int>> edges;
vector<vector<int>> graph;
vector<int> used;

vector<int> height;
vector<int> up;

// Algorithm: Bridges
// Complexity: O(N + M)

vector<int> bridges;

void dfs_bridges(int cur, int p = -1) {
    used[cur] = 1;
    up[cur] = height[cur];
    for (auto &ind : g[cur]) {
        int to = cur ^ edges[ind].ff ^ edges[ind].ss;
        if (to == p) continue;
        if (used[to]) {
            up[cur] = min(up[cur], height[to]);
        }
        else {
            height[to] = height[cur] + 1;
            dfs_bridges(to, cur);
            up[cur] = min(up[cur], up[to]);
            if (up[to] > height[cur])
                bridges.push_back(ind);
        }
    }
}

// Algorithm: ECC
// Complexity: O(N + M)

vector<int> st;

vector<int> add_comp(vector<int> &st, int sz) {
    vector<int> comp;

    while (st.size() > sz) {
        comp.push_back(st.back());
        st.pop_back();
    }

    return comp;
}

vector<vector<int>> ecc;

void dfs_bridges_comps(int cur, int p = -1) {
    used[cur] = 1;
    up[cur] = height[cur];
```

```

for (auto &ind : g[cur]) {
    int to = cur ^ edges[ind].ff ^ edges[ind].ss;
    if (to == p) continue;
    if (used[to]) {
        up[cur] = min(up[cur], height[to]);
    }
    else {
        int sz = st.size();
        st.push_back(to);
        height[to] = height[cur] + 1;
        dfs_bridges_comps(to, cur);
        up[cur] = min(up[cur], up[to]);
        if (up[to] > height[cur])
            ecc.push_back(add_comp(st, sz));
    }
}
}
}

```

## 7.7 Articulation Points

// Theme: Articulation Points And VCC

```

vector<pair<int, int>> edges;
vector<vector<int>> graph;
vector<int> used;

```

```

vector<int> height;
vector<int> up;

```

// Algorithm: Articulation Points  
// Complexity:  $O(N + M)$

```
set<int> art_points;
```

```

void dfs_artics(int cur, int p = -1) {
    used[cur] = 1;
    up[cur] = height[cur];

    int desc_count = 0;

    for (auto &ind : g[cur]) {
        int to = cur ^ edges[ind].ff ^ edges[ind].ss;
        if (to == p) continue;
        if (used[to]) {
            up[cur] = min(up[cur], height[to]);
        }
        else {
            desc_count++;
            height[to] = height[cur] + 1;
            dfs_artics(to, cur);
            up[cur] = min(up[cur], up[to]);
            if (up[to] >= height[cur] && p != -1)
                art_points.insert(cur);
        }
    }

    if (p == -1 && desc_count > 1) {
        art_points.insert(cur);
    }
}

```

// Algorithm: VCC  
// Complexity:  $O(N + M)$

```
vector<vector<int>> vcc;
```

```

void dfs_artics_comps(int cur, int p = -1) {
    used[cur] = 1;
    up[cur] = height[cur];

    for (auto &ind : g[cur]) {
        int to = cur ^ edges[ind].ff ^ edges[ind].ss;
        if (to == p) continue;
        if (used[to]) {
            up[cur] = min(up[cur], height[to]);
            if (height[to] < height[cur]) st.push_back(ind);
        }
        else {
            int sz = st.size();
            st.push_back(ind);
            height[to] = height[cur] + 1;
            dfs_artics_comps(to, cur);
            up[cur] = min(up[cur], up[to]);
            if (up[to] >= height[cur])
                vcc.push_back(add_comp(st, sz));
        }
    }
}

```

```

    }
}
}

```

## 7.8 Kuhn

// Maximum Matching

// Algorithm: Kuhn Algorithm  
// Complexity:  $O(|Left\ Part|^3)$

```

vector<vector<int>> bigraph;
vector<int> used;

```

```
vector<int> mt;
```

```

bool kuhn(int u) {
    if (used[u]) return false;

    used[u] = 1;

```

```

    for (auto &v : bigraph[u]) {
        if (mt[v] == -1 || kuhn(mt[v])) {
            mt[v] = u;
            return true;
        }
    }

```

```

    return false;
}

```

```

int main() {
    ... чтение графа...

```

```

    mt.assign(k, -1);
    for (int v=0; v<n; ++v) {
        used.assign(n, false);
        try_kuhn(v);
    }

```

```

    for (int i=0; i<k; ++i)
        if (mt[i] != -1)
            printf ("%d %d\n", mt[i]+1, i+1);
}

```

## 7.9 Kruskal

```

#include <iostream>
#include <vector>
#include <algorithm>

```

```
using namespace std;
```

```

struct dsu {
    vector<int> p, size;

```

```

    dsu(int n) {
        p.assign(n, 0); size.assign(n, 0);
        for (int i = 0; i < n; i++) {
            p[i] = i;
            size[i] = 1;
        }
    }

```

```

    int get(int v) {
        if (p[v] != v) p[v] = get(p[v]);
        return p[v];
    }

```

```

    void unite(int u, int v) {
        auto x = get(u), y = get(v);
        if (x == y) return;
        if (size[x] > size[y]) swap(x, y);
        p[x] = y; size[y] += size[x];
    }
};

```

```
int sz;
```

```

struct edge {
    long long int u, v, w;

```

```

    edge(long long int uu, long long int vv, long long int
        ww) :u(uu), v(vv), w(ww) {};

};

vector<edge> edges;
vector<vector<int>> graph;

// Algorithm: Kruskal Algorithm
// Complexity: O(M)

vector<edge> mst;

void kruskal() {
    dsu d(sz);

    auto tedges = edges;
    sort(tedges.begin(), tedges.end(), [](edge& e1, edge& e2) {
        return e1.w < e2.w; });

    for (auto& e : tedges) {
        if (d.get(e.u) != d.get(e.v)) {
            mst.push_back(e);
            d.unite(e.u, e.v);
        }
    }
}

int main() {
    long long int n, m, i, j, k, a, b, c;
    cin >> n >> m;
    for (i = 0; i < m; i++) {
        cin >> a >> b >> c;
        a--; b--;
        edge e(a, b, c);
        edges.push_back(e);
    }
    sz = n;
    kruskal();
    long long int ans = 0;
    for (auto it : mst) ans += it.w;
    cout << ans;
}

```

## 7.10 LCA (binary Lifting)

```

// Theme: Lowest Common Ancestor

// Algorithm: Binary Lifting
// Complexity: Preprocessing O(N * log(N)) and request O(log(N))

vector<vector<int>> g;
vector<int> d;
vector<vector<int>> dp;
vector<int> used;

void dfs(int v, int p = -1) {
    if (p == -1) {
        p = v;
        d[v] = 0;
    } else {
        d[v] = d[p] + 1;
    }

    dp[0][v] = p;
    for (int i = 1; i < dp.size(); i++) {
        dp[i][v] = dp[i - 1][dp[i - 1][v]];
    }
    for (int to : g[v]) {
        if (to != p) {
            dfs(to, v);
        }
    }
}

int lca(int a, int b) {
    if (d[a] > d[b]) {
        swap(a, b);
    }

    for (int i = dp.size() - 1; i >= 0; i--) {
        if (d[dp[i][b]] >= d[a]) {
            b = dp[i][b];
        }
    }
}

```

```

    if (a == b) {
        return a;
    }

    for (int i = dp.size() - 1; i >= 0; i--) {
        if (dp[i][a] != dp[i][b]) {
            a = dp[i][a];
            b = dp[i][b];
        }
    }

    return dp[0][a];
}

signed main() {
    int n = 0, m = 0; // n - vertex count, m - requests
    g.resize(n);
    d.resize(n);
    dp.resize(((int)log2(n) + 1));
    used.assign(n, 0);
    for (int i = 0; i < dp.size(); i++) {
        dp[i].resize(n);
    }

    // ...reading graph...

    dfs(0);

    // ...lca(u - 1, v - 1) + 1
}

```

## 7.11 LCA RMQ (seqtree)

```

// Theme: Lowest Common Ancestor

// Algorithm: RMQ (seqtree)
// Complexity: Preprocessing O(N) and request O(log(N))

typedef vector<vector<int>> graph;
typedef vector<int>::const_iterator const_graph_iter;

vector<int> lca_h, lca_dfs_list, lca_first, lca_tree;
vector<char> lca_dfs_used;

void lca_dfs(const graph &g, int v, int h = 1) {
    lca_dfs_used[v] = true;
    lca_h[v] = h;
    lca_dfs_list.push_back(v);
    for (const_graph_iter i = g[v].begin(); i != g[v].end(); ++i)
        if (!lca_dfs_used[*i]) {
            lca_dfs(g, *i, h + 1);
            lca_dfs_list.push_back(v);
        }
}

void lca_build_tree(int i, int l, int r) {
    if (l == r)
        lca_tree[i] = lca_dfs_list[l];
    else {
        int m = (l + r) >> 1;
        lca_build_tree(i + i, l, m);
        lca_build_tree(i + i + 1, m + 1, r);
        if (lca_h[lca_tree[i + i]] < lca_h[lca_tree[i + i + 1]])
            lca_tree[i] = lca_tree[i + i];
        else
            lca_tree[i] = lca_tree[i + i + 1];
    }
}

void lca_prepare(const graph &g, int root) {
    int n = (int)g.size();
    lca_h.resize(n);
    lca_dfs_list.reserve(n * 2);
    lca_dfs_used.assign(n, 0);
    lca_dfs(g, root);
    int m = (int)lca_dfs_list.size();
    lca_tree.assign(lca_dfs_list.size() * 4 + 1, -1);
    lca_build_tree(1, 0, m - 1);
    lca_first.assign(n, -1);
    for (int i = 0; i < m; ++i) {
        int v = lca_dfs_list[i];
        if (lca_first[v] == -1)
            lca_first[v] = i;
    }
}

```



```

}
}

int lca_tree_min(int i, int sl, int sr, int l, int r) {
    if (sl == l && sr == r)
        return lca_tree[i];
    int sm = (sl + sr) >> 1;
    if (r <= sm)
        return lca_tree_min(i + i, sl, sm, l, r);
    if (l > sm)
        return lca_tree_min(i + i + 1, sm + 1, sr, l, r);
    int ans1 = lca_tree_min(i + i, sl, sm, l, sm);
    int ans2 = lca_tree_min(i + i + 1, sm + 1, sr, sm + 1, r);
    ;
    return lca_h[ans1] < lca_h[ans2] ? ans1 : ans2;
}

int lca(int a, int b) {
    int left = lca_first[a],
        right = lca_first[b];
    if (left > right)
        swap(left, right);
    return lca_tree_min(1, 0, (int)lca_dfs_list.size() - 1,
        left, right);
}

signed main() {
    int n = 0, m = 0; // n - vertex count, m - requests
    graph g(n);

    // ...reading graph...

    lca_prepare(g, 0);

    // ...lca(u - 1, v - 1) + 1
}

```

## 7.12 Maximum Flow

```

// Theme: Maximum Flow

int s, t, sz;

vector<edge> edges;
vector<vector<int>> fgraph;

vector<int> used;

// Algorithm: Ford-Fulkerson Algorithm
// Complexity: O(MF)

int dfs_fordfulk(int u, int bound, int flow = INF) {
    if (used[u]) return 0;
    if (u == t) return flow;

    used[u] = 1;

    for (auto &ind : fgraph[u]) {
        auto &e = edges[ind],
            &_e = edges[ind ^ 1];
        int to = e.to, res = e.res();

        if (res < bound) continue;

        int pushed = dfs_fordfulk(to, bound, min(res, flow))
            ;

        if (pushed) {
            e.flow += pushed;
            _e.flow -= pushed;
            return pushed;
        }
    }

    return 0;
}

// Algorithm: Edmonds-Karp Algorithm
// Complexity: O(N(M^2))

vector<int> p;
vector<int> pe;

void augment(int pushed) {
    int cur = t;

```

```

    while (cur != s) {
        auto &e = edges[pe[cur]],
            &_e = edges[pe[cur] ^ 1];
        e.flow += pushed;
        _e.flow -= pushed;
        cur = p[cur];
    }
}

int bfs_edmskarp(int u, int bound) {
    p.assign(sz, -1);
    pe.assign(sz, -1);

    int pushed = 0;

    queue<pair<int, int>> q;
    q.push({ u, INF });

    used[u] = 1;

    while (q.size()) {
        auto [v, f] = q.front(); q.pop();

        for (auto &ind : fgraph[v]) {
            auto &e = edges[ind];
            int to = e.to, res = e.res();

            if (used[to] || res < bound) continue;

            p[to] = v;
            pe[to] = ind;
            used[to] = 1;

            if (to == t) {
                pushed = min(f, res);
                break;
            }

            q.push({ to, min(f, res) });
        }
    }

    if (pushed)
        augment(pushed);

    return pushed;
}

// Algorithm: Dinic Algorithm
// Complexity: O((N^2)M)

vector<int> d;

bool bfs_dinic(int u, int bound) {
    d.assign(sz, INF); d[u] = 0;

    queue<int> q; q.push(u);

    while (q.size()) {
        int v = q.front(); q.pop();

        for (auto &ind : fgraph[v]) {
            auto &e = edges[ind];
            int to = e.to, res = e.res();

            if (d[v] + 1 >= d[to] || res < bound) continue;

            d[to] = d[v] + 1;
            q.push(to);
        }
    }

    return d[t] != INF;
}

vector<int> lst;

int dfs_dinic(int u, int mx = INF) {
    if (u == t) return mx;

    int smf = 0;

    for (int i = lst[u]; i < fgraph[u].size(); i++) {
        int ind = fgraph[u][i];

        auto &e = edges[ind],
            &_e = edges[ind ^ 1];
        int to = e.to, res = e.res();

```

```

    if (d[to] == d[u] + 1 && res) {
        int pushed = dfs_dinic(to, min(res, mx - smf));

        if (pushed) {
            smf += pushed;
            e.flow += pushed;
            _e.flow -= pushed;
        }

        lst[u]++;

        if (smf == mx)
            return smf;
    }

    return smf;
}

int dinic(int u) {
    int pushed = 0;

    for (int bound = 1ll << 30; bound; bound >>= 1) {
        while (true) {
            bool bfs_ok = bfs_dinic(u, bound);
            if (!bfs_ok) break;

            lst.assign(sz, 0);

            while (true) {
                int dfs_pushed = dfs_dinic(u);
                if (!dfs_pushed) break;

                pushed += dfs_pushed;
            }
        }
    }

    return pushed;
}

// Algorithm: Maximum Flow Of Minimum Cost (SPFA)
// Complexity: ...

vector<int> d;
vector<int> p;
vector<int> pe;

void augment(int pushed) {
    int cur = t;
    while (cur != s) {
        auto &e = edges[pe[cur]];
        &_e = edges[pe[cur] ^ 1];
        e.flow += pushed;
        _e.flow -= pushed;
        cur = p[cur];
    }
}

int bfs_spfa(int u, int flow = INF) {
    d.assign(sz, INF); d[u] = 0;
    p.assign(sz, -1);
    pe.assign(sz, -1);

    queue<pair<int, int>> q; q.push({ u, flow });
    vector<int> in_q(sz, 0); in_q[u] = 1;

    int pushed = 0;

    while (q.size()) {
        auto [v, f] = q.front(); q.pop();

        in_q[v] = 0;

        if (v == t) {
            pushed = f;
            break;
        }

        for (auto &ind : fgraph[v]) {
            auto &e = edges[ind];
            int to = e.to, res = e.res(),
                w = e.weight;

            if (d[v] + w >= d[to] || !res) continue;

```

```

            d[to] = d[v] + w;
            p[to] = v;
            pe[to] = ind;

            if (!in_q[to]) {
                in_q[to] = 1;
                q.push({ to, min(f, res) });
            }
        }
    }

    if (pushed)
        augment(pushed);

    return pushed;
}

```

## 7.13 Eulerian Path

```

// Theme: Eulerian Path

int sz;

vector<vector<int>> graph;

// Algorithm: Eulerian Path
// Complexity: O(M)

vector<int> eul;

// 0 - path not exist
// 1 - cycle exists
// 2 - path exists
int euler_path() {
    vector<int> deg(sz);

    for (int i = 0; i < sz; i++)
        for (int j = 0; j < sz; j++)
            deg[i] += g[i][j];

    int v1 = -1, v2 = -1;
    for (int i = 0; i < sz; i++)
        if (deg[i] & 1)
            if (v1 == -1) v1 = i;
            else if (v2 == -1) v2 = i;
            else return 0;

    if (v1 != -1) {
        if (v2 == -1)
            return 0;
        graph[v1][v2]++;
        graph[v2][v1]++;
    }

    stack<int> st;

    for (int i = 0; i < sz; i++) {
        if (deg[i]) {
            st.push(i);
            break;
        }
    }

    while (st.size()) {
        int u = st.top();

        int ind = -1;

        for (int i = 0; i < sz; i++)
            if (graph[u][i]) {
                ind = i;
                break;
            }

        if (ind == -1) {
            eul.push_back(u);
            st.pop();
        }
        else {
            graph[u][ind]--;
            graph[ind][u]--;
            st.push(ind);
        }
    }
}

```

```

int res = 2;
if (v1 != -1) {
    res = 1;

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

    for (int i = 0; i < sz; i++)
        for (int j = 0; j < sz; j++)
            if (graph[i][j])
                return 0;

    return res;
}

```

## 7.14 Eulerian Path Oriented

```

// Theme: Eulerian Path

// Algorithm: Eulerian Path
// Complexity: O(M)

// Algo doesn't validate the path to be correct.
// If the path exists, it will be found.
// Result way has to be reversed after execution.

struct Edge {
    int v, u;
    bool deleted = false;
};

vector<Edge> edges;

vector<deque<int>> graph;
vector<int> way;

void euler(int v, int last = -1) {
    while (!graph[v].empty()) {
        int idx = graph[v].front(); graph[v].pop_front();
        auto& e = edges[idx];
        if (e.deleted) continue;
        e.deleted = true; // If graph is not oriented, also
                          // delete reverse edge[v ^ 1]
        euler(e.u, idx);
    }
    if (last != -1) way.push_back(last);
}

```

## 7.15 Shortest Paths

```

// Theme: Shortest Paths

int sz;

vector<edge> edges;
vector<vector<int>> graph;

// Algorithm: Dijkstra Algorithm
// Complexity: O(M*log(N))

vector<int> d;
vector<int> p;

void dijkstra(int u) {
    d.assign(sz, INF); d[u] = 0;
    p.assign(sz, -1);

    priority_queue<pair<int, int>> q;
    q.push({ 0, u });

    while (q.size()) {
        int dist = -q.top().ff, v = q.top().ss; q.pop();

```

```

        if (dist > d[v]) continue;

        for (auto &ind : graph[v]) {
            int to = v ^ edges[ind].u ^ edges[ind].v,
                w = edges[ind].w;
            if (d[v] + w < d[to]) {
                d[to] = d[v] + w;
                p[to] = v;
                q.push({ -d[to], -to });
            }
        }
    }
}

// Algorithm: Shortest Path Faster Algorithm
// Complexity: ...

vector<int> d;

void bfs_spfa(int u) {
    d.assign(sz, INF); d[u] = 0;

    queue<int> q; q.push(u);
    vector<int> in_q(sz, 0); in_q[u] = 1;

    while (q.size()) {
        auto [v, f] = q.front(); q.pop();

        in_q[v] = 0;

        for (auto &ind : graph[v]) {
            int to = v ^ edges[ind].u ^ edges[ind].v,
                w = edges[ind].w;
            if (d[v] + w < d[to]) {
                d[to] = d[v] + w;
                if (!in_q[to]) {
                    in_q[to] = 1;
                    q.push(to);
                }
            }
        }
    }
}

// Algorithm: Belman-Ford Algorithm
// Complexity: (N*M)

vector<int> d;

void bfa(int u) {
    d.assign(sz, INF); d[u] = 0;

    for (;;) {
        bool any = false;

        for (auto &e : edges) {
            if (d[e.u] != INF && d[e.u] + e.w < d[e.v]) {
                d[e.v] = d[e.u] + e.w;
                any = true;
            }
            if (d[e.v] != INF && d[e.v] + e.w < d[e.u]) {
                d[e.u] = d[e.v] + e.w;
                any = true;
            }
        }

        if (!any) break;
    }
}

// Algorithm: Floyd-Warshall Algorithm
// Complexity: O(N^3)

vector<vector<int>> d;

void fwa() {
    d.assign(sz, vector<int>(sz, INF));

    for (int i = 0; i < sz; i++)
        for (int j = 0; j < sz; j++)
            for (int k = 0; k < sz; k++)
                if (d[i][k] != INF && d[k][j] != INF)
                    d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}

```

## 8 Miscellaneous

### 8.1 Ternary Search

```
// Theme: Ternary Search

// Algorithm: Continuous Search With Golden Ratio
// Complexity:  $O(\log(N))$ 

// Golden Ratio
//  $\Phi = 1.618\dots$ 
double phi = (1 + sqrt(5)) / 2;

double cont_tern_srch(double l, double r) {
    double m1 = l + (r - l) / (1 + phi),
           m2 = r - (r - l) / (1 + phi);

    double f1 = f(m1), f2 = f(m2);

    int count = 200;
    while (count-- > 0) {
        if (f1 < f2) {
            r = m2;
            m2 = m1;
            f2 = f1;
            m1 = l + (r - l) / (1 + phi);
            f1 = f(m1);
        }
        else {
            l = m1;
            m1 = m2;
            f1 = f2;
            m2 = r - (r - l) / (1 + phi);
            f2 = f(m2);
        }
    }

    return f((l + r) / 2);
}

// Algorithm: Discrete Search
// Complexity:  $O(\log(N))$ 

double discr_tern_srch(int l, int r) {
    while (r - l > 2) {
        int m1 = l + (r - l) / 3,
            m2 = r - (r - l) / 3;
        if (f(m1) < f(m2))
            r = m2;
        else
            l = m1;
    }

    return min(f(l), min(f(l + 1), f(r)));
}
```