Contents

1 Template

1.1 Makefile

```
BASIC := -std=c++11 -Wall -Wextra -Wshadow -g -DLOCAL
VERBOSE := -fsanitize=address -fsanitize=undefined -D_GLIBCXX_DEBUG

main: main.cc
	g++ $(BASIC) $(VERBOSE) $< -o $@
```

1.2 vimrc

```
filetype plugin indent on
set nu rnu
set ai ts=4 shiftwidth=4 sts=4 et
set spr sb
set clipboard=unnamed,unnamedplus
```

2 Graph

2.1 Dijkstra

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

using namespace std;

struct Edge {
    int u, v, w;
    Edge(int u_=-1, int v_=-1, int w_=-1) : u(u_), v(v_), w(w_) {}
};

struct Node {
    int u;
    int64_t d;
    Node(int u_, int64_t d_) : u(u_), d(d_) {}
    bool operator<(const Node& o) const {
        return d > o.d; // min-heap
    }
};

struct Graph {
    const int64_t inf = 1e18;
    int n;
    vector<vector<Edge>> adj;
    vector<int64_t> dist;
    vector<Edge> trace; // trace[u]: last edge to get to u from s

    Graph(int n_) : n(n_), adj(n), dist(n, inf),
        trace(n) {}

    void addEdge(int u, int v, int w) {
        adj[u].emplace_back(u, v, w);
    }

    int64_t dijkstra(int s, int t) {
        priority_queue<Node> pq;
        pq.emplace(s, 0);
        dist[s] = 0;

        while (!pq.empty()) {
            Node cur = pq.top(); pq.pop();
            int u = cur.u;
            int64_t d = cur.d;

            if (u == t) return dist[t];
            if (d > dist[u]) continue;

            for (const Edge& e : adj[u]) {
                int v = e.v;
                int w = e.w;
                if (dist[u] + w < dist[v]) {
                    dist[v] = dist[u] + w;
                    trace[v] = e;
                    pq.emplace(v, dist[v]);
                }
            }
        }

        return inf;
    }

    vector<Edge> getShortestPath(int s, int t) {
        assert(dist[t] != inf);
        vector<Edge> path;
        int v = t;
        while (v != s) {
            Edge e = trace[v];
            path.push_back(e);
            v = e.u;
        }
        reverse(path.begin(), path.end());
        return path;
    }
};
```

```cpp
int main() {
    int n, m, s, t;
    cin >> n >> m >> s >> t;

    Graph g(n);

    for (int i = 0; i < m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        g.addEdge(u, v, w);
    }

    int64_t dist = g.dijkstra(s, t);

    if (dist != g.inf) {
        vector<Edge> path = g.getShortestPath(s, t);
        cout << dist << ' ' << path.size() << '\n';
        for (Edge e : path) cout << e.u << ' ' << e.v << '\n';
    } else {
        cout << "-1\n";
    }

    return 0;
}
```

2.2 Strongly Connected Components

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

using namespace std;

// https://judge.yosupo.jp/problem/scc
// Properties:
// - component graph is a DAG
// - traversed graph has the same sccs
// In this implementation, each component is sorted in topological order
struct Graph {
    int n;
    vector<vector<int>> adj;
    vector<vector<int>> adj_t;
    vector<int> mark;
    vector<int> order;
    vector<int> leader;
    vector<vector<int>> components;

    Graph(int n_) : n(n_), adj(n), adj_t(n), mark(n), leader(n) {}

    void addEdge(int u, int v) {
        adj[u].push_back(v);
        adj_t[v].push_back(u);
    }

    void dfsForward(int u) {
        assert(mark[u] == 0);
        mark[u] = 1;

        for (int v : adj[u]) {
            if (mark[v] == 0) {
                dfsForward(v);
            }
        }

        order.push_back(u);
    }

    void dfsBackward(int u, int p) {
        assert(mark[u] == 1);
        mark[u] = 2;
        leader[u] = p;

        for (int v : adj_t[u]) {
            if (mark[v] == 1) {
                dfsBackward(v, p);
            }
        }

        components.back().push_back(u);
    }

    vector<vector<int>> scc() { // Kosaraju's algorithm
        fill(mark.begin(), mark.end(), 0);
        for (int u = 0; u < n; u++) {
            if (mark[u] == 0) {
                dfsForward(u);
            }
        }

        reverse(order.begin(), order.end());

        for (int u : order) {
            if (mark[u] == 1) {
                components.emplace_back();
                dfsBackward(u, u);
            }
        }

        return components;
    }
};

int main() {
    int n, m;
    cin >> n >> m;

    Graph g(n);

    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        g.addEdge(u, v);
    }

    vector<vector<int>> components = g.scc();

    cout << components.size() << '\n';

    for (vector<int>& comp : components) {
        cout << comp.size() << ' ';
        for (int u : comp) {
            cout << u << ' ';
        }
        cout << '\n';
    }

    return 0;
}
```

3 Structures

3.1 Disjoint Set/Union-Find/Disjoint-Set-Union (DSU))

```cpp
#pragma once

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

```cpp

using namespace std;

struct DSU {
    int n;
    vector<int> p;
    vector<int> d;

    DSU(int n_): n(n_), p(n), d(n, 0) {
        for (int i = 0; i < n; i++) p[i] = i;
    }

    int get(int u) {
        while (u != p[u]) u = p[u]; return u;
    }

    bool merge(int u, int v) {
        u = get(u);
        v = get(v);
        if (u == v) return false;
        if (d[u] < d[v]) {
            p[u] = v;
        } else if (d[u] > d[v]) {
            p[v] = u;
        } else {
            p[u] = v;
            d[v]++;
        }
        return true;
    }
};
```

3.2 Segment Tree

```cpp
#pragma once

#include <bits/stdc++.h>

using namespace std;

template <typename T>
using BinOp = function<T(T, T)>;

template <typename T>
struct SegmentTree {
    struct Node {
        int from;
        int to;
        T val;
        T lazy;
        bool is_lazy;
    };

    int n;
    vector<Node> t;
    T dlazy;
    BinOp<T> merge;
    T dquery;

    SegmentTree(vector<int>& a, T dlazy_, BinOp<T> merge,
    ↪  T dquery_) :
        n(a.size()), t(n * 4), dlazy(dlazy_),
        ↪  merge(merge), dquery(dquery_) {
        build(a, 0, 0, n - 1);
    }

    virtual void apply(int u, T delta) = 0;
    virtual void pushDown(int u) = 0;

    inline int left(int u)  { return 2 * u + 1; }
    inline int right(int u) { return 2 * u + 2; }

    void build(vector<int>& a, int u, int from, int to) {
        if (from == to) {
            t[u] = Node({from, to, a[from], dlazy,
            ↪  false});
            return;
        }

        int l = left(u);
        int r = right(u);
        int mid = (from + to) / 2;
        build(a, l, from, mid);
        build(a, r, mid + 1, to);
        T val = merge(t[l].val, t[r].val);
        t[u] = Node({from, to, val, dlazy, false});
    }

    T query(int from, int to, int u=0) {
        if (from <= t[u].from && t[u].to <= to) return
        ↪  t[u].val;
        if (to < t[u].from || t[u].to < from) return
        ↪  dquery;
        pushDown(u);
        return merge(query(from, to, left(u)),
                     query(from, to, right(u)));
    }

    void update(int from, int to, T delta, int u=0) {
        if (from > to) return;

        if (from == t[u].from && to == t[u].to) {
            apply(u, delta);
            return;
        }

        pushDown(u);
        int l = left(u);
        int r = right(u);
        int mid = (t[u].from + t[u].to) / 2;
        update(from, min(to, mid), delta, l);
        update(max(from, mid + 1), to, delta, r);
        t[u].val = merge(t[l].val, t[r].val);
    }
};

template <typename T>
struct SegmentAssignUpdate : public SegmentTree<T> {
    SegmentAssignUpdate(vector<int>& a, BinOp<T> merge_,
    ↪  int dquery_) :
        SegmentTree<T>(a, 0, merge_, dquery_) {}

    virtual void apply(int u, T delta) {
        auto& t = this->t;
        t[u].val = delta;
        t[u].is_lazy = true;
    }

    virtual void pushDown(int u) {
        auto& t = this->t;
        int l = this->left(u);
        int r = this->right(u);
        if (t[u].is_lazy) {
            t[l].val = t[r].val = t[u].val;
            t[l].is_lazy = t[r].is_lazy = true;
            t[u].is_lazy = false;
        }
    }
};
```

```cpp

template <typename T>
struct SegmentAddUpdate : SegmentTree<T> {
    SegmentAddUpdate(vector<int>& a, int dlazy_, BinOp<T> merge_, int dquery_) :
            SegmentTree<T>(a, dlazy_, merge_, dquery_) {}

    virtual void apply(int u, T delta) {
        auto& t = this->t;
        t[u].val += delta;
        t[u].lazy += delta;
    }

    virtual void pushDown(int u) {
        auto& t = this->t;
        int l = this->left(u);
        int r = this->right(u);
        t[l].val += t[u].lazy;
        t[l].lazy += t[u].lazy;
        t[r].val += t[u].lazy;
        t[r].lazy += t[u].lazy;
        t[u].lazy = 0;
    }
};
```

4 Maths

4.1 Modular Arithmetic

```cpp
// **Really important note**: inputs of the modAdd, modSub, and modMul
// functions must all be normalized (within the range [0..mod - 1]) before use

#pragma once

#include <bits/stdc++.h>

using namespace std;

int modAdd(int a, int b, int mod) {
    a += b;
    if (a >= mod) a -= mod;
    return a;
}

int modSub(int a, int b, int mod) {
    a -= b;
    if (a < 0) a += mod;
    return a;
}

int modMul(int a, int b, int mod) {
    int64_t res = (int64_t) a * b;
    return (int) (res % mod);
}

int64_t binPow(int64_t a, int64_t x) {
    int64_t res = 1;
    while (x) {
        if (x & 1) res *= a;
        a *= a;
        x >>= 1;
    }
    return res;
}

int64_t modPow(int64_t a, int64_t x, int mod) {
    int res = 1;
    while (x) {
        if (x & 1) res = modMul(res, a, mod);
        a = modMul(a, a, mod);
        x >>= 1;
    }
    return res;
}
```

4.2 Modnum

```cpp
#pragma once

#include <bits/stdc++.h>
#include "mod.hpp"
#include "mod_inverse.hpp"

using namespace std;

template <typename T, int md>
struct Modnum {
    using M = Modnum;
    T v;
    Modnum(int64_t v_=0) : v(fix(v_)) {}

    T fix(int64_t x) {
        if (x < -md || x > 2 * md) x %= md;
        if (x >= md) x -= md;
        if (x < 0) x += md;
        return x;
    }

    M operator-()    { return M(-v); };
    M operator+(M o) { return M(v + o.v); }
    M operator-(M o) { return M(v - o.v); }
    M operator*(M o) { return M(fix((int64_t) v * o.v)); }
    M operator/(M o) { return *this * modInv(o.v, md); }
    M pow(int64_t x) {
        M a(v);
        M res(1);
        while (x) {
            if (x & 1) res = res * a;
            a = a * a;
            x >>= 1;
        }
        return res;
    }

    friend istream& operator>>(istream& is, M& o) {
        is >> o.v; o.v = o.fix(o.v); return is;
    }
    friend ostream& operator<<(ostream& os, const M& o) {
        return os << o.v;
    }

    friend T abs(const M& m) { if (m.v < 0) return -m.v; return m.v; }
};
```

4.3 Sieve of Eratosthenes

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

using namespace std;

/// Sieve of Eratosthenes
/// Benchmark: 3314 ms/188.74 Mib for N = 5 * 1e8
/// Credit: KTH's notebook
```

```cpp
constexpr int MAX_N = (int) 5 * 1e8;
bitset<MAX_N + 1> is_prime;
vector<int> primes;

void sieve(int N) {
    is_prime.set();
    is_prime[0] = is_prime[1] = 0;

    for (int i = 4; i <= N; i += 2) is_prime[i] = 0;

    for (int i = 3; i * i <= N; i += 2) {
        if (!is_prime[i]) continue;
        for (int j = i * i; j <= N; j += i * 2) {
            is_prime[j] = 0;
        }
    }

    for (int i = 2; i <= N; i++) {
        if (is_prime[i]) primes.push_back(i);
    }
}

// https://judge.yosupo.jp/problem/enumerate_primes
int main() {
    int N, a, b;
    cin >> N >> a >> b;
    sieve(N);
    int num_primes = primes.size();
    vector<int> res;

    for (int j = 0; a * j + b < num_primes; j++) {
        res.push_back(primes[a * j + b]);
    }

    cout << num_primes << ' ' << res.size() << '\n';

    for (int p : res) {
        cout << p << ' ';
    }
    cout << '\n';
}
```

4.4 Primality Test

```cpp
// Simple primality test

#pragma once

#include <bits/stdc++.h>

template <typename T>
bool isPrime(T x) {
    for (T d = 2; d * d <= x; d++) {
        if (x % d == 0) return false;
    }
    return true;
}
```

4.5 Euclidean Algorithm

```cpp
#pragma once

#include <bits/stdc++.h>

using namespace std;

template <typename T>
T gcd(T a, T b) {
    if (a < b) swap(a, b);
    while (b != 0) {
        int r = a % b;
        a = b;
        b = r;
    }
    return a;
}

template <typename T>
int64_t lcm(T a, T b) {
    return (int64_t) a / gcd(a, b) * b;
}
```

4.6 Extended Euclidean Algorithm

```cpp
#pragma once

#include "mod.hpp"

// This solves the equation $ax + by = gcd(a, b)$
// Input: $a$, $b$
// Output: $g$ (returned), $x$, $y$ (passed by ref)
int64_t extGcd(int64_t a, int64_t b, int64_t& x, int64_t& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    int64_t x1, y1;
    int64_t g = extGcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    assert(g == 1);
    return g;
}
```

4.7 Euler's Totient Function

```cpp
#pragma once

#include <bits/stdc++.h>

using namespace std;

// Euler's totient function
// $\phi(i)$ = number of coprime numbers of n in the range $[1..n]$
// Multiplicative property: $\phi(a * b) = phi(a) * phi(b)$
// Complexity: $O(\sqrt(n))$
int eulerPhi(int n) {
    int res = n;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            while (n % i == 0) {
                n /= i;
            }
            res -= res / i;
        }
    }
    if (n > 1) {
        res -= res / n;
    }
    return res;
}

// Complexity: $O(n \log \log (n))$
```

```cpp
vector<int> eulerPhiN(int n) {
    vector<int> phi(n + 1);
    phi[0] = 0;
    phi[1] = 1;

    for (int i = 2; i <= n; i++) phi[i] = i;

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

    return phi;
}
```

4.8 Matrix

```cpp
#pragma once

#include <bits/stdc++.h>

using namespace std;

template <typename T>
struct vec2d : public vector<vector<T>> {
vec2d(int n=0, int m=0, T t=T())
        : vector<vector<T>>(n, vector<T>(m, t)) {}
};

template <typename T>
struct Matrix : vec2d<T> {
    int n;

    Matrix(int n_, T t=T()) : vec2d<T>(n_, n_, t), n(n_) {}

    Matrix operator+(const Matrix& o) const {
        assert(n == o.n);
        const Matrix& a = *this;
        Matrix res(n);

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                res[i][j] = a[i][j] + o[i][j];
            }
        }

        return res;
    }

    Matrix operator-(const Matrix& o) const {
        assert(n == o.n);
        const Matrix& a = *this;
        Matrix res(n);

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                res[i][j] = a[i][j] - o[i][j];
            }
        }

        return res;
    }

    Matrix operator*(const Matrix& o) const {
        assert(n == o.n);
        const Matrix& a = *this;
        Matrix res(n, 0);

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                for (int k = 0; k < n; k++) {
                    res[i][j] = res[i][j] + a[i][k] * o[k][j];
                }
            }
        }
        return res;
    }

    void identity() {
        Matrix& a = *this;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (i == j) a[i][j] = 1;
                else a[i][j] = 0;
            }
        }
    }

    // Gauss method. Complexity: O(n^3)
    friend T determinant(const Matrix& mat) {
        int n = mat.n;
        Matrix a(n);

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

        const double EPS = 1E-9;
        T det = 1;

        for (int i = 0; i < n; ++i) {
            int k = i;

            for (int j= i + 1; j < n; j++) {
                if (abs(a[j][i]) > abs(a[k][i])) {
                    k = j;
                }
            }

            if (abs(a[k][i]) < EPS) {
                det = 0;
                break;
            }

            swap(a[i], a[k]);

            if (i != k) det = -det;

            det = det * a[i][i];

            for (int j = i + 1; j < n; j++) {
                a[i][j] = a[i][j] / a[i][i];
            }

            for (int j = 0; j < n; j++) {
                if (j != i && abs(a[j][i]) > EPS) {
                    for (int k = i + 1; k < n; k++) {
                        a[j][k] = a[j][k] - a[i][k] * a[j][i];
                    }
                }
            }
        }
```

```
        }

        return det;
    }
};
```

5 Strings

5.1 Trie

```
#pragma once

#include <bits/stdc++.h>

using namespace std;

struct Trie {
    const int ALPHA = 26;
    vector<vector<int>> trie;
    vector<int> eow;

    int ord(char c) { return c - 'a'; }

    Trie() {
        trie.emplace_back(ALPHA, -1);
        eow.push_back(0);
    }

    void add(const string& word) {
        int node = 0;

        for (char c : word) {
            int x = ord(c);

            if (trie[node][x] == -1) {
                trie[node][x] = trie.size();
                trie.emplace_back(ALPHA, -1);
                eow.push_back(0);
            }

            node = trie[node][x];
            eow[node]++;
        }
    }
};
```

5.2 Z function

```
#pragma once

#include <bits/stdc++.h>

using namespace std;

// z[i]: length of the longest common prefix between s and
// its substring starting at i
vector<int> zFunction(const string& s) {
    int n = s.length();
    vector<int> z(n);
    z[0] = n;
    int l = 0;
    int r = 0;

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

5.3 Suffix Array

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

using namespace std;

// sa[i] = the starting index of the ith suffix (starting at 0)
// sorted in lexicographic order
vector<int> suffix_array(const string& s_, int alpha=256) {
    string s = s_ + '\0';
    int n = s.size();
    vector<int> p(n);
    vector<int> cnt(max(alpha, n), 0);

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

    vector<int> g(n);
    g[p[0]] = 0;

    for (int i = 1; i < n; i++) {
        g[p[i]] = g[p[i - 1]] + (s[p[i]] != s[p[i - 1]]);
    }

    vector<int> pn(n);
    vector<int> gn(n);

    for (int len = 1; len < n; len <<= 1) {
        for (int i = 0; i < n; i++) {
            pn[i] = p[i] - len; // transfer the pos from second to pair
            if (pn[i] < 0) pn[i] += n; // cyclic
        }

        int num_groups = g[p[n - 1]] + 1;
        fill(cnt.begin(), cnt.begin() + num_groups, 0);

        // Radix sort
        for (int i = 0; i < n; i++) cnt[g[pn[i]]]++;
        for (int i = 1; i < num_groups; i++) cnt[i] += cnt[i - 1];
        for (int i = n - 1; i >= 0; i--) p[--cnt[g[pn[i]]]] = pn[i];
        gn[p[0]] = 0;

        for (int i = 1; i < n; i++) {
            pair<int, int> prev, cur;
            prev.first = g[p[i - 1]];
            cur.first = g[p[i]];
            prev.second = g[p[i - 1] + len - (p[i - 1] + len >= n ? n : 0)];
            cur.second = g[p[i] + len - (p[i] + len >= n ? n : 0)];
            gn[p[i]] = gn[p[i - 1]] + (cur != prev);
```

```
        }
        g.swap(gn);
    }
    p.erase(p.begin());
    return p;
}
```

6 Flows

6.1 Dinic Max Flow

```
#pragma once

#include <bits/stdc++.h>

using namespace std;

/// Dinic algorithm for max flow
/// This versionshould work on flow graph with float capacities
/// Time complexity: O(|V|^2|E|)

template <typename T>
struct FlowEdge {
    int u, v;
    T c, f;

    FlowEdge(int _u, int _v, T _c, T _f) :
            u(_u), v(_v), c(_c), f(_f) {}
};

template <typename T>
struct Dinic {
    static constexpr T inf = numeric_limits<T>::max();
    static constexpr T eps = (T) 1e-9;
    int n;
    int s, t;
    vector<vector<int>> adj; // stores indices of edges
    vector<int> level;       // shortest distance from source
    vector<int> ptr;         // points to the next edge which can be used
    vector<FlowEdge<T>> edges;

    Dinic(int _n, int _s, int _t)
            : n(_n), s(_s), t(_t), adj(_n), level(_n), ptr(_n) {}

    void addEdge(int u, int v, int c, int rc=0) {
        int eid = (int) edges.size();
        adj[u].push_back(eid);
        adj[v].push_back(eid + 1);
        edges.emplace_back(u, v, c, 0);
        edges.emplace_back(v, u, rc, 0);
    }

    bool bfs() {
        fill(level.begin(), level.end(), -1);
        level[s] = 0;
        queue<int> q;
        q.push(s);

        while (!q.empty()) {
            int u = q.front();
            q.pop();

            for (int eid : adj[u]) {
                const auto& e = edges[eid];
                if (e.c - e.f <= eps || level[e.v] != -1) continue;
                level[e.v] = level[u] + 1;
                q.push(e.v);
            }
        }

        return level[t] != -1;
    }

    T dfs(int u, T flow) {
        if (u == t) return flow;

        for (int& j = ptr[u]; j < (int) adj[u].size(); j++) {
            int eid = adj[u][j];
            const auto& e = edges[eid];
            if (e.c - e.f > eps && level[e.v] == level[u] + 1) {
                T df = dfs(e.v, min(e.c - e.f, flow));
                if (df > eps) {
                    edges[eid].f += df;
                    edges[eid ^ 1].f -= df;
                    return df;
                }
            }
        }

        return 0;
    }

    T maxFlow() {
        T f = 0;

        while (bfs()) {
            fill(ptr.begin(), ptr.end(), 0);
            T total_df = 0;
            while (true) {
                T df = dfs(s, inf);
                if (df <= eps) break;
                total_df += df;
            }
            if (total_df <= eps) break;
            f += total_df;
        }

        return f;
    }
};
```

7 Matching

7.1 Hopcroft-Karp Bipartite Matching

```
#pragma once

#include <bits/stdc++.h>

using namespace std;

#pragma once

// Bipartite matching. Vertices from both halves start from 0
// Time complexity: O(sqrt(|V|)|E|)
struct HopcroftKarp {
    const int INF = (int) 1e9;
    int nu;
    int nv;
    vector<vector<int>> adj;
```

```
    vector<int> layer;
    vector<int> u_mate;
    vector<int> v_mate;

    HopcroftKarp(int nu, int nv) : nu(nu), nv(nv) {
        adj.resize(nu);
        layer.resize(nu);
        u_mate.resize(nu, -1);
        v_mate.resize(nv, -1);
    }

    void addEdge(int u, int v) {
        adj[u].push_back(v);
    }

    bool bfs() {
        // Find all possible augmenting paths
        queue<int> q;

        for (int u = 0; u < nu; u++) {
            // Consider only unmatched edges
            if (u_mate[u] == -1) {
                layer[u] = 0;
                q.push(u);
            } else {
                layer[u] = INF;
            }
        }

        bool has_path = false;

        while (!q.empty()) {
            int u = q.front();
            q.pop();

            for (int &v : adj[u]) {
                if (v_mate[v] == -1) {
                    has_path = true;
                } else if (layer[v_mate[v]] == INF) {
                    layer[v_mate[v]] = layer[u] + 1;
                    q.push(v_mate[v]);
                }
            }
        }

        return has_path;
    }

    bool dfs(int u) {
        if (layer[u] == INF) return false;

        for (int v : adj[u]) {
            if ((v_mate[v] == -1) ||
                (layer[v_mate[v]] == layer[u] + 1 &&
                ↪ dfs(v_mate[v]))) {
                v_mate[v] = u;
                u_mate[u] = v;
                return true;
            }
        }

        return false;
    }

    vector<pair<int, int>> maxMatching() {
        int matching = 0;

        while (bfs()) { // there is at least 1 augmenting
        ↪ path
            for (int u = 0; u < nu; u++) {
                if (u_mate[u] == -1 && dfs(u)) {
                    ++matching;
                }
            }
        }

        vector<pair<int, int>> res;

        for (int u = 0; u < nu; u++) {
            if (u_mate[u] == -1) continue;
            res.emplace_back(u, u_mate[u]);
        }
        assert(res.size() == matching);
        return res;
    }
};
```

8 Geometry

8.1 Utility

```
#pragma once

#include <bits/stdc++.h>

using namespace std;

const double PI = acos(-1);

template <typename T>
int sgn(T x) {
    if (x > 0) return 1;
    if (x < 0) return -1;
    return 0;
}

int inc(int i, int n, int by=1) {
    i += by;
    if (i >= n) i -= n;
    return i;
}

double degToRad(double d) {
    return d * PI / 180.0;
}

double radToDeg(double r) {
    return r * 180.0 / PI;
}
```

8.2 Point

```
#pragma once

#include <bits/stdc++.h>
#include "geoutil.hpp"

using namespace std;


template<typename T>
struct Point {
    using P = Point;
    T x, y;

    Point(T x_ = 0, T y_ = 0) : x(x_), y(y_) {}
    P operator+(const P &o) const { return P(x + o.x, y +
    ↪ o.y); }
```

```
    P operator-(const P &o) const { return P(x - o.x, y - o.y); }
    P operator*(T d) const { return P(x * d, y * d); }
    P operator/(T d) const { return P(x / d, y / d); }
    T dot(P o) const { return x * o.x + y * o.y; }
    T cross(P o) const { return x * o.y - y * o.x; }
    T abs2() const { return x * x + y * y; }
    long double abs() const { return sqrt((long double) abs2()); }
    double angle() const { return atan2(y, x); } // [−π, π]
    P unit() const { return *this / abs(); } // makes abs()=1
    P perp() const { return P(-y, x); } // rotates +π/2

    P rotate(double a) const { // ccw
        return P(x * cos(a) - y * sin(a), x * sin(a) + y * cos(a));
    }

    friend istream &operator>>(istream &is, P &p) {
        return is >> p.x >> p.y;
    }

    friend ostream &operator<<(ostream &os, P &p) {
        return os << "(" << p.x << ", " << p.y << ")";
    }

    // position of c relative to a->b
    // > 0: c is on the left of a->b
    friend T orient(P a, P b, P c) {
        return (b - a).cross(c - a);
    }

    // Check if u⃗ and v⃗ are parallel
    // (u⃗ = cv⃗) where c ∈ R)
    friend bool parallel(P u, P v) {
        return u.cross(v) == 0;
    }

    // Check if point p lies on the segment ab
    friend bool onSegment(P a, P b, P p) {
        return orient(a, b, p) == 0 &&
               min(a.x, b.x) <= p.x &&
               max(a.x, b.x) >= p.x &&
               min(a.y, b.y) <= p.y &&
               max(a.y, b.y) >= p.y;
    }

    friend bool boundingBox(P p1, P q1, P p2, P q2) {
        if (max(p1.x, q1.x) < min(p2.x, q2.x)) return true;
        if (max(p1.y, q1.y) < min(p2.y, q2.y)) return true;
        if (max(p2.x, q2.x) < min(p1.x, q1.x)) return true;
        if (max(p2.x, q2.x) < min(p1.x, q1.x)) return true;
        return false;
    }

    friend bool intersect(P p1, P p2, P p3, P p4) {
        // Check if two segments are parallel
        if (parallel(p2 - p1, p4 - p3)) {
            // Check if 4 ps are colinear
            if (!parallel(p2 - p1, p3 - p1)) return false;
            if (boundingBox(p1, p2, p3, p4)) return false;
            return true;
        }

        // check if one line is completely on one side of the other
        for (int i = 0; i < 2; i++) {
            if (sgn(orient(p1, p2, p3)) == sgn(orient(p1, p2, p4))
                && sgn(orient(p1, p2, p3)) != 0) {
                return false;
            }
            swap(p1, p3);
            swap(p2, p4);
        }
        return true;
    }

    // Check if p is in ∠bac (including the rays)
    friend bool inAngle(P a, P b, P c, P p) {
        assert(orient(a, b, c) != 0);
        if (orient(a, b, c) < 0) swap(b, c);
        return orient(a, b, p) >= 0 && orient(a, c, p) <= 0;
    }

    // Angle ∠bac (+/-)
    friend double directedAngle(P a, P b, P c) {
        if (orient(a, b, c) >= 0) {
            return (b - a).angle(c - a);
        }
        return 2 * PI - (b - a).angle(c - a);
    }
};
```

8.3 Polygon

```
#pragma once

#include <bits/stdc++.h>
#include "point.hpp"
#include "geoutil.hpp"
#include "../maths/euclidean.hpp"

using namespace std;

template <typename T>
struct Polygon {
    using P = Point<T>;

    int n = 0;
    vector<P> ps;
    Polygon() : n(0) {}
    Polygon(vector<P>& ps) : n(ps.size()), ps(ps) {}

    void add(P p) {
        ps.push_back(p);
        n++;
    }

    int64_t twiceArea() {
        int64_t area = 0;
        for (int i = 0; i < n; i++) {
            P p1 = ps[i];
            P p2 = ps[inc(i, n)];
            area += p1.cross(p2);
        }
        return abs(area);
    }

    double area() {
```

```
    return twiceArea() / 2.0;
}

int64_t boundaryLattice() {
    int64_t res = 0;
    for (int i = 0; i < n; i++) {
        int j = i + 1; if (j == n) j = 0;
        P p1 = ps[i];
        P p2 = ps[j];
        P v = p2 - p1;
        res += gcd(abs(v.x), abs(v.y));
    }
    return res;
}

int64_t interiorLattice() {
    return (twiceArea() - boundaryLattice()) / 2 + 1;
}

bool isConvex() {
    int pos = 0;
    int neg = 0;

    for (int i = 0; i < n; i++) {
        P p1 = ps[i];
        P p2 = ps[inc(i, n, 1)];
        P p3 = ps[inc(i, n, 2)];
        int o = orient(p1, p2, p3);
        if (o > 0) pos = 1;
        if (o < 1) neg = 1;
    }

    return pos ^ neg;
}

// -1: outside; 1: inside; 0: on boundary
int vsPoint(P r) {
    int crossing = 0;
    for (int i = 0; i < n; i++) {
        P p1 = ps[i];
        P p2 = ps[inc(i, n)];
        if (onSegment(p1, p2, r)) {
            return 0;
        }
        if (((p2.y >= r.y) - (p1.y >= r.y)) *
        ↪  orient(r, p1, p2) > 0) {
            crossing++;
        }
    }
    if (crossing & 1) return 1;
    return -1;
}
};

template <typename T>
Polygon<T> convexHull(vector<Point<T>> points) {
    using P = Point<T>;

    sort(points.begin(), points.end(),
         [](const P& p1, const P& p2) {
             if (p1.x == p2.x) return p1.y < p2.y;
             return p1.x < p2.x;
         });

    vector<P> hull;

    for (int step = 0; step < 2; step++) {
        int s = hull.size();
        for (const P& c : points) {
            while ((int) hull.size() - s >= 2) {
                P a = hull.end()[-2];
                P b = hull.end()[-1];
                // <= if points on the edges are
                ↪  accepted, < otherwise
                if (orient(a, b, c) <= 0) break;
                hull.pop_back();
            }
            hull.push_back(c);
        }
        hull.pop_back();
        reverse(points.begin(), points.end());
    }

    return Polygon<T>(hull);
}
```

9 C++ STL

9.1 vector

Underlying implementation: dynamic array

Method	Complexity
size_t size()	$O(1)$
void push_back(T v)	$O(1)$
void emplace_back(Args args...)	$O(1)$
void pop_back()	$O(1)$
T back()	$O(1)$
void erase(iterator position)	$O(n)$

- Resize (values in vector stay unchanged): v.resize(n)
- Resize and fill: v.assign(n, val)
- Fill: fill(v.begin(), v.end(), val)
- Reverse: reverse(v.begin(), v.end())
- Pythonic get element backwards:
  - v.end()[−1]: last element
  - v.end()[−2]: second-last element
- Sort ():

```
// by default: non-decreasing, v must be of
↪  comparator type
sort(v.begin(), v.end());
// custom comparator
sort(v.begin(), v.end(), [](const Obj& o1, const
↪  Obj& o2) {
  return o1.x < o2.x;
});
```

9.2 set

Condition: must be of a comparable type (define the < operator).
Underlying implementation: self-balancing BST

Method	Complexity
size_t size()	$O(1)$
void insert(T v)	$O(1)$
void emplace(Args args...)	$O(1)$
iterator find(T v)	$O(\log(n))$
void erase(iterator position)	$O(\log(n))$

- Check if an element v is in set s: if (s.find(v) != s.end())
- Get minimum element: *(m.begin())
- Get maximum element: *(m.rbegin())

9.3 map

Condition: **key** must be of a comparable type (define the < operator).

Underlying implementation: self-balancing BST

Method	Complexity
size_t size()	$O(1)$
void insert(pair<K, V> keyvalpair)	$O(1)$
void emplace(K key, V value)	$O(1)$
iterator find(T v)	$O(\log(n))$
void erase(iterator position)	$O(\log(n))$

- Check if a key k is in map m: if (m.find(k) != m.end())
- Get value of key k in map m: m[k] or m.find(k)->second
- Get minimum key-value pair: *(m.begin())
- Get key of minimum pair: m.begin()->first
- Get value of minimum pair: m.begin()->second
- Get maximum key-value pair: *(m.rbegin())
- Get key of maximum pair: m.rbegin()->first
- Get value of maximum pair: m.rbegin()->second

9.4 unordered_set and unordered_map

Underlying implementation: hash table

Note: stay always from these unless you know what you are doing. There are scenarios where you think these can be faster than set and map, but either:

- The speed-up it will be negligible
- It will actually be unexpectedly slower

Operations: pretty much share the same interface with set and map, except for things that require order.

9.5 pair

Lexicographically comparable

9.6 string

- Mutable: s[0] = 'a' is OK.
- Concatenation:
  - s += 'a' takes $O(1)$!
  - s += t takes $O(length(t))$
- Substring:
  - s.substr(i) returns suffix starting from i
  - s.substr(i, 3) returns suffix starting from i of maximum length 3 (can be shorter if reaches end)

9.7 Other useful utilities

min(x, y), max(x, y), swap(x, y)