

Contents

1	Template	1
1.1	Makefile	1
1.2	vimrc	1
2	Graph	1
2.1	Dijkstra	1
3	Maths	2
3.1	Modular Arithmetic	2
3.2	Modnum	3
3.3	Sieve of Eratosthenes	3
3.4	Primality Test	4
3.5	Euclidean Algorithm	4
3.6	Extended Euclidean Algorithm	4
3.7	Euler's Totient Function	4
4	Strings	5
4.1	Trie	5
4.2	Z function	5
4.3	Suffix Array	6
5	Flows	6
5.1	Dinic Max Flow	6
6	Matching	8
6.1	Hopcroft-Karp Bipartite Matching	8
7	Geometry	9
7.1	Utility	9
7.2	Point	9
7.3	Polygon	10

1 Template

1.1 Makefile

```
1 BASIC := -std=c++11 -Wall -Wextra -Wshadow -g -DLOCAL
2 VERBOSE := -fsanitize=address -fsanitize=undefined -D_GLIBCXX_DEBUG
3
4 main: main.cc
5      g++ $(BASIC) $(VERBOSE) $< -o $@
```

1.2 vimrc

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

2 Graph

2.1 Dijkstra

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 struct Edge {
6     int u, v, w;
7     Edge(int u_=-1, int v_=-1, int w_=-1) : u(u_), v(v_), w(w_) {}
8 };
9
10 struct Node {
11     int u;
12     int64_t d;
13     Node(int u_, int64_t d_) : u(u_), d(d_) {}
14     bool operator<(const Node& o) const {
15         return d > o.d; // min-heap
16     }
17 };
18
19 struct Graph {
20     const int64_t inf = 1e18;
21     int n;
22     vector<vector<Edge>> adj;
23     vector<int64_t> dist;
24     vector<Edge> trace; // trace[u]: last edge to get to u from s
25
26     Graph(int n_) : n(n_), adj(n), dist(n, inf),
27         trace(n) {}
28
29     void addEdge(int u, int v, int w) {
30         adj[u].emplace_back(u, v, w);
31     }
32
33     int64_t dijkstra(int s, int t) {
34         priority_queue<Node> pq;
35         pq.emplace(s, 0);
36         dist[s] = 0;
37
38         while (!pq.empty()) {
39             Node cur = pq.top(); pq.pop();
40             int u = cur.u;
41             int64_t d = cur.d;
42
43             if (u == t) return dist[t];
```

```

44         if (d > dist[u]) continue;
45
46         for (const Edge& e : adj[u]) {
47             int v = e.v;
48             int w = e.w;
49             if (dist[u] + w < dist[v]) {
50                 dist[v] = dist[u] + w;
51                 trace[v] = e;
52                 pq.emplace(v, dist[v]);
53             }
54         }
55     }
56
57     return inf;
58 }
59
60 vector<Edge> getShortestPath(int s, int t) {
61     assert(dist[t] != inf);
62     vector<Edge> path;
63     int v = t;
64     while (v != s) {
65         Edge e = trace[v];
66         path.push_back(e);
67         v = e.u;
68     }
69     reverse(path.begin(), path.end());
70     return path;
71 }
72 };
73
74
75 int main() {
76     int n, m, s, t;
77     cin >> n >> m >> s >> t;
78
79     Graph g(n);
80
81     for (int i = 0; i < m; i++) {
82         int u, v, w;
83         cin >> u >> v >> w;
84         g.addEdge(u, v, w);
85     }
86
87     int64_t dist = g.dijkstra(s, t);
88
89     if (dist != g.inf) {
90         vector<Edge> path = g.getShortestPath(s, t);

```

```

91         cout << dist << ' ' << path.size() << '\n';
92         for (Edge e : path) cout << e.u << ' ' << e.v << '\n';
93     } else {
94         cout << "-1\n";
95     }
96
97     return 0;
98 }

```

3 Maths

3.1 Modular Arithmetic

```

1  // **Really important note**: inputs of the modAdd, modSub, and modMul
2  // functions must all be normalized (within the range [0..mod - 1]) before use
3
4  #pragma once
5
6  #include <bits/stdc++.h>
7
8  using namespace std;
9
10 int modAdd(int a, int b, int mod) {
11     a += b;
12     if (a >= mod) a -= mod;
13     return a;
14 }
15
16 int modSub(int a, int b, int mod) {
17     a -= b;
18     if (a < 0) a += mod;
19     return a;
20 }
21
22 int modMul(int a, int b, int mod) {
23     int64_t res = (int64_t) a * b;
24     return (int) (res % mod);
25 }
26
27 int64_t binPow(int64_t a, int64_t x) {
28     int64_t res = 1;
29     while (x) {
30         if (x & 1) res *= a;
31         a *= a;
32         x >>= 1;
33     }

```

```

34     return res;
35 }
36
37 int64_t modPow(int64_t a, int64_t x, int mod) {
38     int res = 1;
39     while (x) {
40         if (x & 1) res = modMul(res, a, mod);
41         a = modMul(a, a, mod);
42         x >>= 1;
43     }
44     return res;
45 }

```

3.2 Modnum

```

1  #pragma once
2
3  #include <bits/stdc++.h>
4  #include "mod.hpp"
5
6  using namespace std;
7
8  template <typename T, int md>
9  struct Modnum {
10     using M = Modnum;
11     T v;
12     Modnum(T _v=0) : v(fix(_v)) {}
13
14     T fix(int64_t x) {
15         if (x < -md || x > 2 * md) x %= md;
16         if (x >= md) x -= md;
17         if (x < 0) x += md;
18         return x;
19     }
20
21     M operator+(M o) { return M(v + o.v); }
22     M operator-(M o) { return M(v - o.v); }
23     M operator*(M o) { return M(fix((int64_t) v * o.v)); }
24     M operator/(M o) {
25         return *this * modInv(o.v, md);
26     }
27     M pow(int64_t x) {
28         M a(v);
29         M res(1);
30         while (x) {
31             if (x & 1) res = res * a;

```

```

32         a = a * a;
33         x >>= 1;
34     }
35     return res;
36 }
37 friend istream& operator>>(istream& is, M& o) {
38     is >> o.v; o.v = o.fix(o.v); return is;
39 }
40 friend ostream& operator<<(ostream& os, const M& o) {
41     return os << o.v;
42 }
43 };

```

3.3 Sieve of Eratosthenes

```

1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  /// Sieve of Eratosthenes
6  /// Benchmark: 3314 ms/188.74 Mib for N = 5 * 1e8
7  /// Credit: KTH's notebook
8  constexpr int MAX_N = (int) 5 * 1e8;
9  bitset<MAX_N + 1> is_prime;
10 vector<int> primes;
11
12 void sieve(int N) {
13     is_prime.set();
14     is_prime[0] = is_prime[1] = 0;
15
16     for (int i = 4; i <= N; i += 2) is_prime[i] = 0;
17
18     for (int i = 3; i * i <= N; i += 2) {
19         if (!is_prime[i]) continue;
20         for (int j = i * i; j <= N; j += i * 2) {
21             is_prime[j] = 0;
22         }
23     }
24
25     for (int i = 2; i <= N; i++) {
26         if (is_prime[i]) primes.push_back(i);
27     }
28 }
29
30 // https://judge.yosupo.jp/problem/enumerate_primes
31 int main() {

```

```
32     int N, a, b;
33     cin >> N >> a >> b;
34     sieve(N);
35     int num_primes = primes.size();
36     vector<int> res;
37
38     for (int j = 0; a * j + b < num_primes; j++) {
39         res.push_back(primes[a * j + b]);
40     }
41
42     cout << num_primes << ' ' << res.size() << '\n';
43
44     for (int p : res) {
45         cout << p << ' ';
46     }
47     cout << '\n';
48 }
```

3.4 Primality Test

```
1 // Simple primality test
2
3 #pragma once
4
5 #include <bits/stdc++.h>
6
7 template <typename T>
8 bool isPrime(T x) {
9     for (T d = 2; d * d <= x; d++) {
10         if (x % d == 0) return false;
11     }
12     return true;
13 }
```

3.5 Euclidean Algorithm

```
1 #pragma once
2
3 #include <bits/stdc++.h>
4
5 using namespace std;
6
7 template <typename T>
8 T gcd(T a, T b) {
9     if (a < b) swap(a, b);
```

```
10     while (b != 0) {
11         int r = a % b;
12         a = b;
13         b = r;
14     }
15     return a;
16 }
17
18 template <typename T>
19 int64_t lcm(T a, T b) {
20     return (int64_t) a / gcd(a, b) * b;
21 }
```

3.6 Extended Euclidean Algorithm

```
1 #pragma once
2
3 #include "mod.hpp"
4
5 // This solves the equation  $ax + by = \gcd(a, b)$ 
6 // Input: a, b
7 // Output: g (returned), x, y (passed by ref)
8 int64_t extGcd(int64_t a, int64_t b, int64_t& x, int64_t& y) {
9     if (b == 0) {
10         x = 1;
11         y = 0;
12         return a;
13     }
14     int64_t x1, y1;
15     int64_t g = extGcd(b, a % b, x1, y1);
16     x = y1;
17     y = x1 - y1 * (a / b);
18     assert(g == 1);
19     return g;
20 }
```

3.7 Euler's Totient Function

```
1 #pragma once
2
3 #include <bits/stdc++.h>
4
5 using namespace std;
6
```

```

7 // Euler's totient function
8 //  $\phi(i)$  = number of coprime numbers of  $n$  in the range  $[1..n]$ 
9 // Multiplicative property:  $\phi(a * b) = \phi(a) * \phi(b)$ 
10 // Complexity:  $O(\sqrt{n})$ 
11 int eulerPhi(int n) {
12     int res = n;
13     for (int i = 2; i * i <= n; i++) {
14         if (n % i == 0) {
15             while (n % i == 0) {
16                 n /= i;
17             }
18             res -= res / i;
19         }
20     }
21     if (n > 1) {
22         res -= res / n;
23     }
24     return res;
25 }
26
27 // Complexity:  $O(n \log \log(n))$ 
28 vector<int> eulerPhiN(int n) {
29     vector<int> phi(n + 1);
30     phi[0] = 0;
31     phi[1] = 1;
32
33     for (int i = 2; i <= n; i++) phi[i] = i;
34
35     for (int i = 2; i <= n; i++) {
36         if (phi[i] == i) {
37             for (int j = i; j <= n; j += i) {
38                 phi[j] -= phi[j] / i;
39             }
40         }
41     }
42
43     return phi;
44 }

```

4 Strings

4.1 Trie

```

1 #pragma once
2
3 #include <bits/stdc++.h>

```

```

4
5 using namespace std;
6
7 struct Trie {
8     const int ALPHA = 26;
9     vector<vector<int>> trie;
10    vector<int> eow;
11
12    int ord(char c) { return c - 'a'; }
13
14    Trie() {
15        trie.emplace_back(ALPHA, -1);
16        eow.push_back(0);
17    }
18
19    void add(const string& word) {
20        int node = 0;
21
22        for (char c : word) {
23            int x = ord(c);
24
25            if (trie[node][x] == -1) {
26                trie[node][x] = trie.size();
27                trie.emplace_back(ALPHA, -1);
28                eow.push_back(0);
29            }
30
31            node = trie[node][x];
32            eow[node]++;
33        }
34    }
35 };

```

4.2 Z function

```

1 #pragma once
2
3 #include <bits/stdc++.h>
4
5 using namespace std;
6
7 // z[i]: length of the longest common prefix between  $s$  and
8 // its substring starting at  $i$ 
9 vector<int> zFunction(const string& s) {
10     int n = s.length();
11     vector<int> z(n);

```

```

12     z[0] = n;
13     int l = 0;
14     int r = 0;
15
16     for (int i = 1; i < n; i++) {
17         if (i <= r) {
18             z[i] = min(z[i - 1], r - i + 1);
19         }
20         while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
21             z[i]++;
22         }
23         if (i + z[i] - 1 > r) {
24             l = i;
25             r = i + z[i] - 1;
26         }
27     }
28
29     return z;
30 }

```

4.3 Suffix Array

```

1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  // sa[i] = the starting index of the ith suffix (starting at 0)
6  // sorted in lexicographic order
7  vector<int> suffix_array(const string& s_, int alpha=256) {
8      string s = s_ + '\0';
9      int n = s.size();
10     vector<int> p(n);
11     vector<int> cnt(max(alpha, n), 0);
12
13     for (int i = 0; i < n; i++) cnt[s[i]]++;
14     for (int i = 1; i < alpha; i++) cnt[i] += cnt[i - 1];
15     for (int i = 0; i < n; i++) p[--cnt[s[i]]] = i;
16
17     vector<int> g(n);
18     g[p[0]] = 0;
19
20     for (int i = 1; i < n; i++) {
21         g[p[i]] = g[p[i - 1]] + (s[p[i]] != s[p[i - 1]]);
22     }
23
24     vector<int> pn(n);

```

```

25     vector<int> gn(n);
26
27     for (int len = 1; len < n; len <= 1) {
28         for (int i = 0; i < n; i++) {
29             pn[i] = p[i] - len; // transfer the pos from second to pair
30             if (pn[i] < 0) pn[i] += n; // cyclic
31         }
32
33         int num_groups = g[p[n - 1]] + 1;
34         fill(cnt.begin(), cnt.begin() + num_groups, 0);
35
36         // Radix sort
37         for (int i = 0; i < n; i++) cnt[g[pn[i]]]++;
38         for (int i = 1; i < num_groups; i++) cnt[i] += cnt[i - 1];
39         for (int i = n - 1; i >= 0; i--) p[--cnt[g[pn[i]]]] = pn[i];
40         gn[p[0]] = 0;
41
42         for (int i = 1; i < n; i++) {
43             pair<int, int> prev, cur;
44             prev.first = g[p[i - 1]];
45             cur.first = g[p[i]];
46             prev.second = g[p[i - 1]] + len - (p[i - 1] + len >= n ? n : 0);
47             cur.second = g[p[i]] + len - (p[i] + len >= n ? n : 0);
48             gn[p[i]] = gn[p[i - 1]] + (cur != prev);
49         }
50         g.swap(gn);
51     }
52     p.erase(p.begin());
53     return p;
54 }

```

5 Flows

5.1 Dinic Max Flow

```

1  #pragma once
2
3  #include <bits/stdc++.h>
4
5  using namespace std;
6
7  /// Dinic algorithm for max flow
8  /// This versionshould work on flow graph with float capacities
9  /// Time complexity: O(|V|^2|E|)
10
11 template <typename T>

```

```

12 struct FlowEdge {
13     int u, v;
14     T c, f;
15
16     FlowEdge(int _u, int _v, T _c, T _f) :
17         u(_u), v(_v), c(_c), f(_f) {}
18 };
19
20 template <typename T>
21 struct Dinic {
22     static constexpr T inf = numeric_limits<T>::max();
23     static constexpr T eps = (T) 1e-9;
24     int n;
25     int s, t;
26     vector<vector<int>> adj; // stores indices of edges
27     vector<int> level;      // shortest distance from source
28     vector<int> ptr;        // points to the next edge which can be used
29     vector<FlowEdge<T>> edges;
30
31     Dinic(int _n, int _s, int _t)
32         : n(_n), s(_s), t(_t), adj(_n), level(_n), ptr(_n) {}
33
34     void addEdge(int u, int v, int c, int rc=0) {
35         int eid = (int) edges.size();
36         adj[u].push_back(eid);
37         adj[v].push_back(eid + 1);
38         edges.emplace_back(u, v, c, 0);
39         edges.emplace_back(v, u, rc, 0);
40     }
41
42     bool bfs() {
43         fill(level.begin(), level.end(), -1);
44         level[s] = 0;
45         queue<int> q;
46         q.push(s);
47
48         while (!q.empty()) {
49             int u = q.front();
50             q.pop();
51
52             for (int eid : adj[u]) {
53                 const auto& e = edges[eid];
54                 if (e.c - e.f <= eps || level[e.v] != -1) continue;
55                 level[e.v] = level[u] + 1;
56                 q.push(e.v);
57             }
58         }

```

```

59
60         return level[t] != -1;
61     }
62
63     T dfs(int u, T flow) {
64         if (u == t) return flow;
65
66         for (int& j = ptr[u]; j < (int) adj[u].size(); j++) {
67             int eid = adj[u][j];
68             const auto& e = edges[eid];
69             if (e.c - e.f > eps && level[e.v] == level[u] + 1) {
70                 T df = dfs(e.v, min(e.c - e.f, flow));
71                 if (df > eps) {
72                     edges[eid].f += df;
73                     edges[eid ^ 1].f -= df;
74                     return df;
75                 }
76             }
77         }
78
79         return 0;
80     }
81
82     T maxFlow() {
83         T f = 0;
84
85         while (bfs()) {
86             fill(ptr.begin(), ptr.end(), 0);
87             T total_df = 0;
88             while (true) {
89                 T df = dfs(s, inf);
90                 if (df <= eps) break;
91                 total_df += df;
92             }
93             if (total_df <= eps) break;
94             f += total_df;
95         }
96
97         return f;
98     }
99 };

```

6 Matching

6.1 Hopcroft-Karp Bipartite Matching

```
1  #pragma once
2
3  #include <bits/stdc++.h>
4
5  using namespace std;
6
7  #pragma once
8
9  // Bipartite matching. Vertices from both halves start from 0
10 // Time complexity:  $O(\sqrt{|V|}|E|)$ 
11 struct HopcroftKarp {
12     const int INF = (int) 1e9;
13     int nu;
14     int nv;
15     vector<vector<int>> adj;
16     vector<int> layer;
17     vector<int> u_mate;
18     vector<int> v_mate;
19
20     HopcroftKarp(int nu, int nv) : nu(nu), nv(nv) {
21         adj.resize(nu);
22         layer.resize(nu);
23         u_mate.resize(nu, -1);
24         v_mate.resize(nv, -1);
25     }
26
27     void addEdge(int u, int v) {
28         adj[u].push_back(v);
29     }
30
31     bool bfs() {
32         // Find all possible augmenting paths
33         queue<int> q;
34
35         for (int u = 0; u < nu; u++) {
36             // Consider only unmatched edges
37             if (u_mate[u] == -1) {
38                 layer[u] = 0;
39                 q.push(u);
40             } else {
41                 layer[u] = INF;
42             }
43         }
```

```
44
45         bool has_path = false;
46
47         while (!q.empty()) {
48             int u = q.front();
49             q.pop();
50
51             for (int &v : adj[u]) {
52                 if (v_mate[v] == -1) {
53                     has_path = true;
54                 } else if (layer[v_mate[v]] == INF) {
55                     layer[v_mate[v]] = layer[u] + 1;
56                     q.push(v_mate[v]);
57                 }
58             }
59         }
60
61         return has_path;
62     }
63
64     bool dfs(int u) {
65         if (layer[u] == INF) return false;
66
67         for (int v : adj[u]) {
68             if ((v_mate[v] == -1) ||
69                 (layer[v_mate[v]] == layer[u] + 1 && dfs(v_mate[v]))) {
70                 v_mate[v] = u;
71                 u_mate[u] = v;
72                 return true;
73             }
74         }
75
76         return false;
77     }
78
79     vector<pair<int, int>> maxMatching() {
80         int matching = 0;
81
82         while (bfs()) { // there is at least 1 augmenting path
83             for (int u = 0; u < nu; u++) {
84                 if (u_mate[u] == -1 && dfs(u)) {
85                     ++matching;
86                 }
87             }
88         }
89
90         vector<pair<int, int>> res;
```



```

91         for (int u = 0; u < nu; u++) {
92             if (u_mate[u] == -1) continue;
93             res.emplace_back(u, u_mate[u]);
94         }
95         assert(res.size() == matching);
96         return res;
97     }
98 };
99

```

7 Geometry

7.1 Utility

```

1  #pragma once
2
3  #include <bits/stdc++.h>
4
5  using namespace std;
6
7  const double PI = acos(-1);
8
9  template <typename T>
10 int sgn(T x) {
11     if (x > 0) return 1;
12     if (x < 0) return -1;
13     return 0;
14 }
15
16 int inc(int i, int n, int by=1) {
17     i += by;
18     if (i >= n) i -= n;
19     return i;
20 }
21
22 double degToRad(double d) {
23     return d * PI / 180.0;
24 }
25
26 double radToDeg(double r) {
27     return r * 180.0 / PI;
28 }

```

7.2 Point

```

1  #pragma once
2
3  #include <bits/stdc++.h>
4  #include "geoutil.hpp"
5
6  using namespace std;
7
8
9  template<typename T>
10 struct Point {
11     using P = Point;
12     T x, y;
13
14     Point(T x_ = 0, T y_ = 0) : x(x_), y(y_) {}
15     P operator+(const P &o) const { return P(x + o.x, y + o.y); }
16     P operator-(const P &o) const { return P(x - o.x, y - o.y); }
17     P operator*(T d) const { return P(x * d, y * d); }
18     P operator/(T d) const { return P(x / d, y / d); }
19     T dot(P o) const { return x * o.x + y * o.y; }
20     T cross(P o) const { return x * o.y - y * o.x; }
21     T abs2() const { return x * x + y * y; }
22     long double abs() const { return sqrt((long double) abs2()); }
23     double angle() const { return atan2(y, x); } // [-π, π]
24     P unit() const { return *this / abs(); } // makes abs()=1
25     P perp() const { return P(-y, x); } // rotates +π/2
26
27     P rotate(double a) const { // ccw
28         return P(x * cos(a) - y * sin(a), x * sin(a) + y * cos(a));
29     }
30
31     friend istream &operator>>(istream &is, P &p) {
32         return is >> p.x >> p.y;
33     }
34
35     friend ostream &operator<<(ostream &os, P &p) {
36         return os << "(" << p.x << ", " << p.y << ")";
37     }
38
39     // position of c relative to a->b
40     // > 0: c is on the left of a->b
41     friend T orient(P a, P b, P c) {
42         return (b - a).cross(c - a);
43     }
44
45     // Check if  $\vec{u}$  and  $\vec{v}$  are parallel

```

```

46 // ( $\vec{u} = c\vec{v}$ ) where  $c \in R$ 
47 friend bool parallel(P u, P v) {
48     return u.cross(v) == 0;
49 }
50
51 // Check if point  $p$  lies on the segment  $ab$ 
52 friend bool onSegment(P a, P b, P p) {
53     return orient(a, b, p) == 0 &&
54         min(a.x, b.x) <= p.x &&
55         max(a.x, b.x) >= p.x &&
56         min(a.y, b.y) <= p.y &&
57         max(a.y, b.y) >= p.y;
58 }
59
60 friend bool boundingBox(P p1, P q1, P p2, P q2) {
61     if (max(p1.x, q1.x) < min(p2.x, q2.x)) return true;
62     if (max(p1.y, q1.y) < min(p2.y, q2.y)) return true;
63     if (max(p2.x, q2.x) < min(p1.x, q1.x)) return true;
64     if (max(p2.y, q2.y) < min(p1.y, q1.y)) return true;
65     return false;
66 }
67
68 friend bool intersect(P p1, P p2, P p3, P p4) {
69     // Check if two segments are parallel
70     if (parallel(p2 - p1, p4 - p3)) {
71         // Check if 4 ps are colinear
72         if (!parallel(p2 - p1, p3 - p1)) return false;
73         if (boundingBox(p1, p2, p3, p4)) return false;
74         return true;
75     }
76
77     // check if one line is completely on one side of the other
78     for (int i = 0; i < 2; i++) {
79         if (sgn(orient(p1, p2, p3)) == sgn(orient(p1, p2, p4))
80             && sgn(orient(p1, p2, p3)) != 0) {
81             return false;
82         }
83         swap(p1, p3);
84         swap(p2, p4);
85     }
86     return true;
87 }
88
89 // Check if  $p$  is in  $\angle bac$  (including the rays)
90 friend bool inAngle(P a, P b, P c, P p) {
91     assert(orient(a, b, c) != 0);
92     if (orient(a, b, c) < 0) swap(b, c);

```

```

93     return orient(a, b, p) >= 0 && orient(a, c, p) <= 0;
94 }
95
96 // Angle  $\angle bac$  (+/-)
97 friend double directedAngle(P a, P b, P c) {
98     if (orient(a, b, c) >= 0) {
99         return (b - a).angle(c - a);
100     }
101     return 2 * PI - (b - a).angle(c - a);
102 }
103 };

```

7.3 Polygon

```

1  #pragma once
2
3  #include <bits/stdc++.h>
4  #include "point.hpp"
5  #include "geoutil.hpp"
6  #include "../maths/euclidean.hpp"
7
8  using namespace std;
9
10 template <typename T>
11 struct Polygon {
12     using P = Point<T>;
13
14     int n = 0;
15     vector<P> ps;
16     Polygon() : n(0) {}
17     Polygon(vector<P>& ps) : n(ps.size()), ps(ps) {}
18
19     void add(P p) {
20         ps.push_back(p);
21         n++;
22     }
23
24     int64_t twiceArea() {
25         int64_t area = 0;
26         for (int i = 0; i < n; i++) {
27             P p1 = ps[i];
28             P p2 = ps[inc(i, n)];
29             area += p1.cross(p2);
30         }
31         return abs(area);
32     }

```

```

33
34 double area() {
35     return twiceArea() / 2.0;
36 }
37
38 int64_t boundaryLattice() {
39     int64_t res = 0;
40     for (int i = 0; i < n; i++) {
41         int j = i + 1; if (j == n) j = 0;
42         P p1 = ps[i];
43         P p2 = ps[j];
44         P v = p2 - p1;
45         res += gcd(abs(v.x), abs(v.y));
46     }
47     return res;
48 }
49
50 int64_t interiorLattice() {
51     return (twiceArea() - boundaryLattice()) / 2 + 1;
52 }
53
54 bool isConvex() {
55     int pos = 0;
56     int neg = 0;
57
58     for (int i = 0; i < n; i++) {
59         P p1 = ps[i];
60         P p2 = ps[inc(i, n, 1)];
61         P p3 = ps[inc(i, n, 2)];
62         int o = orient(p1, p2, p3);
63         if (o > 0) pos = 1;
64         if (o < 1) neg = 1;
65     }
66
67     return pos ^ neg;
68 }
69
70 // -1: outside; 1: inside; 0: on boundary
71 int vsPoint(P r) {
72     int crossing = 0;
73     for (int i = 0; i < n; i++) {
74         P p1 = ps[i];
75         P p2 = ps[inc(i, n)];
76         if (onSegment(p1, p2, r)) {
77             return 0;
78         }
79         if (((p2.y >= r.y) - (p1.y >= r.y)) * orient(r, p1, p2) > 0) {

```

```

80             crossing++;
81         }
82     }
83     if (crossing & 1) return 1;
84     return -1;
85 }
86 };
87
88 template <typename T>
89 Polygon<T> convexHull(vector<Point<T>> points) {
90     using P = Point<T>;
91
92     sort(points.begin(), points.end(),
93          [](const P& p1, const P& p2) {
94             if (p1.x == p2.x) return p1.y < p2.y;
95             return p1.x < p2.x;
96         });
97
98     vector<P> hull;
99
100     for (int step = 0; step < 2; step++) {
101         int s = hull.size();
102         for (const P& c : points) {
103             while ((int) hull.size() - s >= 2) {
104                 P a = hull.end()[-2];
105                 P b = hull.end()[-1];
106                 // <= if points on the edges are accepted, < otherwise
107                 if (orient(a, b, c) <= 0) break;
108                 hull.pop_back();
109             }
110             hull.push_back(c);
111         }
112         hull.pop_back();
113         reverse(points.begin(), points.end());
114     }
115
116     return Polygon<T>(hull);
117 }

```
