

## Contents

<b>1</b>	<b>Template</b>	<b>1</b>
1.1	Makefile	1
1.2	vimrc	1
<b>2</b>	<b>Graph</b>	<b>1</b>
2.1	Dijkstra	1
2.2	Strongly Connected Components	2
<b>3</b>	<b>Structures</b>	<b>2</b>
3.1	Disjoint Set/Union-Find/Disjoint-Set-Union (DSU)	2
3.2	Segment Tree	3
<b>4</b>	<b>Maths</b>	<b>4</b>
4.1	Modular Arithmetic	4
4.2	Modnum	4
4.3	Sieve of Eratosthenes	4
4.4	Primality Test	5
4.5	Euclidean Algorithm	5
4.6	Extended Euclidean Algorithm	5
4.7	Euler's Totient Function	5
4.8	Matrix	6
<b>5</b>	<b>Strings</b>	<b>7</b>
5.1	Trie	7
5.2	Z function	7
5.3	Suffix Array	7
<b>6</b>	<b>Flows</b>	<b>8</b>
6.1	Dinic Max Flow	8
<b>7</b>	<b>Matching</b>	<b>8</b>
7.1	Hopcroft-Karp Bipartite Matching	8
<b>8</b>	<b>Geometry</b>	<b>9</b>
8.1	Utility	9
8.2	Point	9
8.3	Polygon	10
<b>9</b>	<b>C++ STL</b>	<b>11</b>
9.1	vector	11
9.2	set	11
9.3	map	12
9.4	unordered_set and unordered_map	12
9.5	pair	12
9.6	string	12
9.7	Other useful utilities	12

## 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 rn
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_),
8         ↳ w(w_) {}
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
25         ↳ u from s
26
27     Graph(int n_) : n(n_), adj(n), dist(n, inf),
28         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 }

```

## 2.2 Strongly Connected Components

```

1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  // https://judge.yosupo.jp/problem/scc
6  // Properties:
7  // - component graph is a DAG
8  // - traversed graph has the same sccs
9  // In this implementation, each component is sorted in
   ↪ topological order
10 struct Graph {
11     int n;
12     vector<vector<int>> adj;
13     vector<vector<int>> adj_t;
14     vector<int> mark;
15     vector<int> order;
16     vector<int> leader;
17     vector<vector<int>> components;

```

```

18
19     Graph(int n_) : n(n_), adj(n), adj_t(n),
20         mark(n), leader(n) {}
21
22     void addEdge(int u, int v) {
23         adj[u].push_back(v);
24         adj_t[v].push_back(u);
25     }
26
27     void dfsForward(int u) {
28         assert(mark[u] == 0);
29         mark[u] = 1;
30
31         for (int v : adj[u]) {
32             if (mark[v] == 0) {
33                 dfsForward(v);
34             }
35         }
36
37         order.push_back(u);
38     }
39
40     void dfsBackward(int u, int p) {
41         assert(mark[u] == 1);
42         mark[u] = 2;
43         leader[u] = p;
44
45         for (int v : adj_t[u]) {
46             if (mark[v] == 1) {
47                 dfsBackward(v, p);
48             }
49         }
50
51         components.back().push_back(u);
52     }
53
54     vector<vector<int>> scc() { // Kosaraju's algorithm
55         fill(mark.begin(), mark.end(), 0);
56         for (int u = 0; u < n; u++) {
57             if (mark[u] == 0) {
58                 dfsForward(u);
59             }
60         }
61
62         reverse(order.begin(), order.end());
63
64         for (int u : order) {

```

```

65             if (mark[u] == 1) {
66                 components.emplace_back();
67                 dfsBackward(u, u);
68             }
69         }
70
71         return components;
72     }
73 };
74
75 int main() {
76     int n, m;
77     cin >> n >> m;
78
79     Graph g(n);
80
81     for (int i = 0; i < m; i++) {
82         int u, v;
83         cin >> u >> v;
84         g.addEdge(u, v);
85     }
86
87     vector<vector<int>> components = g.scc();
88
89     cout << components.size() << '\n';
90
91     for (vector<int>& comp : components) {
92         cout << comp.size() << ' ';
93         for (int u : comp) {
94             cout << u << ' ';
95         }
96         cout << '\n';
97     }
98
99     return 0;
100 }

```

## 3 Structures

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

```

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

```

```

4
5 using namespace std;
6
7 struct DSU {
8     int n;
9     vector<int> p;
10    vector<int> d;
11
12    DSU(int n_): n(n_), p(n), d(n, 0) {
13        for (int i = 0; i < n; i++) p[i] = i;
14    }
15
16    int get(int u) {
17        while (u != p[u]) u = p[u]; return u;
18    }
19
20    bool merge(int u, int v) {
21        u = get(u);
22        v = get(v);
23        if (u == v) return false;
24        if (d[u] < d[v]) {
25            p[u] = v;
26        } else if (d[u] > d[v]) {
27            p[v] = u;
28        } else {
29            p[u] = v;
30            d[v]++;
31        }
32        return true;
33    }
34 };

```

## 3.2 Segment Tree

```

1 #pragma once
2
3 #include <bits/stdc++.h>
4
5 using namespace std;
6
7 template <typename T>
8 using BinOp = function<T(T, T)>;
9
10 template <typename T>
11 struct SegmentTree {
12     struct Node {

```

```

13         int from;
14         int to;
15         T val;
16         T lazy;
17         bool is_lazy;
18     };
19
20     int n;
21     vector<Node> t;
22     T dlazy;
23     BinOp<T> merge;
24     T dquery;
25
26     SegmentTree(vector<int>& a, T dlazy_, BinOp<T> merge,
27         ↪ T dquery_) :
28         n(a.size()), t(n * 4), dlazy(dlazy_),
29         ↪ merge(merge), dquery(dquery_) {
30         build(a, 0, 0, n - 1);
31     }
32
33     virtual void apply(int u, T delta) = 0;
34     virtual void pushDown(int u) = 0;
35
36     inline int left(int u) { return 2 * u + 1; }
37     inline int right(int u) { return 2 * u + 2; }
38
39     void build(vector<int>& a, int u, int from, int to) {
40         if (from == to) {
41             t[u] = Node({from, to, a[from], dlazy,
42                 ↪ false});
43             return;
44         }
45
46         int l = left(u);
47         int r = right(u);
48         int mid = (from + to) / 2;
49         build(a, l, from, mid);
50         build(a, r, mid + 1, to);
51         T val = merge(t[l].val, t[r].val);
52         t[u] = Node({from, to, val, dlazy, false});
53     }
54
55     T query(int from, int to, int u=0) {
56         if (from <= t[u].from && t[u].to <= to) return
57             ↪ t[u].val;
58         if (to < t[u].from || t[u].to < from) return
59             ↪ dquery;

```

```

60         pushDown(u);
61         return merge(query(from, to, left(u)),
62             query(from, to, right(u)));
63     }
64
65     void update(int from, int to, T delta, int u=0) {
66         if (from > to) return;
67
68         if (from == t[u].from && to == t[u].to) {
69             apply(u, delta);
70             return;
71         }
72
73         pushDown(u);
74         int l = left(u);
75         int r = right(u);
76         int mid = (t[u].from + t[u].to) / 2;
77         update(from, min(to, mid), delta, l);
78         update(max(from, mid + 1), to, delta, r);
79         t[u].val = merge(t[l].val, t[r].val);
80     }
81
82     };
83
84     template <typename T>
85     struct SegmentAssignUpdate : public SegmentTree<T> {
86         SegmentAssignUpdate(vector<int>& a, BinOp<T> merge_,
87             ↪ int dquery_) :
88             SegmentTree<T>(a, 0, merge_, dquery_) {}
89
90         virtual void apply(int u, T delta) {
91             auto& t = this->t;
92             t[u].val = delta;
93             t[u].is_lazy = true;
94         }
95
96         virtual void pushDown(int u) {
97             auto& t = this->t;
98             int l = this->left(u);
99             int r = this->right(u);
100             if (t[u].is_lazy) {
101                 t[l].val = t[r].val = t[u].val;
102                 t[l].is_lazy = t[r].is_lazy = true;
103                 t[u].is_lazy = false;
104             }
105         }
106     };

```

```

100
101 template <typename T>
102 struct SegmentAddUpdate : SegmentTree<T> {
103     SegmentAddUpdate(vector<int>& a, int lazy_, BinOp<T>
104         ↪ merge_, int dquery_) :
105         SegmentTree<T>(a, lazy_, merge_, dquery_) {}
106
107     virtual void apply(int u, T delta) {
108         auto& t = this->t;
109         t[u].val += delta;
110         t[u].lazy += delta;
111     }
112
113     virtual void pushDown(int u) {
114         auto& t = this->t;
115         int l = this->left(u);
116         int r = this->right(u);
117         t[l].val += t[u].lazy;
118         t[l].lazy += t[u].lazy;
119         t[r].val += t[u].lazy;
120         t[r].lazy += t[u].lazy;
121         t[u].lazy = 0;
122     };

```

---

## 4 Maths

### 4.1 Modular Arithmetic

```

1 // **Really important note**: inputs of the modAdd,
2 ↪ modSub, and modMul
3 // functions must all be normalized (within the range
4 ↪ [0..mod - 1]) before use
5
6 #pragma once
7
8 #include <bits/stdc++.h>
9
10 using namespace std;
11
12 int modAdd(int a, int b, int mod) {
13     a += b;
14     if (a >= mod) a -= mod;
15     return a;
16 }

```

```

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 }

```

---

### 4.2 Modnum

```

1 #pragma once
2
3 #include <bits/stdc++.h>
4 #include "mod.hpp"
5 #include "mod_inverse.hpp"
6
7 using namespace std;
8
9 template <typename T, int md>
10 struct Modnum {
11     using M = Modnum;
12     T v;
13     Modnum(int64_t v_=0) : v(fix(v_)) {}

```

```

14
15     T fix(int64_t x) {
16         if (x < -md || x > 2 * md) x %= md;
17         if (x >= md) x -= md;
18         if (x < 0) x += md;
19         return x;
20     }
21
22     M operator-() { return M(-v); };
23     M operator+(M o) { return M(v + o.v); }
24     M operator-(M o) { return M(v - o.v); }
25     M operator*(M o) { return M(fix((int64_t) v * o.v));
26     ↪ }
27     M operator/(M o) { return *this * modInv(o.v, md); }
28     M pow(int64_t x) {
29         M a(v);
30         M res(1);
31         while (x) {
32             if (x & 1) res = res * a;
33             a = a * a;
34             x >>= 1;
35         }
36         return res;
37     }
38     friend istream& operator>>(istream& is, M& o) {
39         is >> o.v; o.v = o.fix(o.v); return is;
40     }
41     friend ostream& operator<<(ostream& os, const M& o) {
42         return os << o.v;
43     }
44
45     friend T abs(const M& m) { if (m.v < 0) return -m.v;
46     ↪ return m.v; }
47 };

```

---

### 4.3 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 }

```

## 4.4 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 }

```

## 4.5 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 }

```

## 4.6 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 }

```

## 4.7 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 }

```

## 4.8 Matrix

```

1  #pragma once
2
3  #include <bits/stdc++.h>
4
5  using namespace std;
6
7  template <typename T>
8  struct vec2d : public vector<vector<T>> {
9      vec2d(int n=0, int m=0, T t=T())
10         : vector<vector<T>>(n, vector<T>(m, t)) {}
11 };
12
13 template <typename T>
14 struct Matrix : vec2d<T> {
15     int n;
16
17     Matrix(int n_, T t=T()) : vec2d<T>(n_, n_, t), n(n_)
18         ↪ {}
19
20     Matrix operator+(const Matrix& o) const {
21         assert(n == o.n);
22         const Matrix& a = *this;
23         Matrix res(n);
24
25         for (int i = 0; i < n; i++) {
26             for (int j = 0; j < n; j++) {

```

```

26                 res[i][j] = a[i][j] + o[i][j];
27             }
28         }
29
30         return res;
31     }
32
33     Matrix operator-(const Matrix& o) const {
34         assert(n == o.n);
35         const Matrix& a = *this;
36         Matrix res(n);
37
38         for (int i = 0; i < n; i++) {
39             for (int j = 0; j < n; j++) {
40                 res[i][j] = a[i][j] - o[i][j];
41             }
42         }
43
44         return res;
45     }
46
47     Matrix operator*(const Matrix& o) const {
48         assert(n == o.n);
49         const Matrix& a = *this;
50         Matrix res(n, 0);
51
52         for (int i = 0; i < n; i++) {
53             for (int j = 0; j < n; j++) {
54                 for (int k = 0; k < n; k++) {
55                     res[i][j] = res[i][j] + a[i][k] *
56                         ↪ o[k][j];
57                 }
58             }
59             return res;
60         }
61
62         void identity() {
63             Matrix& a = *this;
64             for (int i = 0; i < n; i++) {
65                 for (int j = 0; j < n; j++) {
66                     if (i == j) a[i][j] = 1;
67                     else a[i][j] = 0;
68                 }
69             }
70         }
71

```

```

72         // Gauss method. Complexity: O(n^3)
73         friend T determinant(const Matrix& mat) {
74             int n = mat.n;
75             Matrix a(n);
76
77             for (int i = 0; i < n; i++) {
78                 for (int j = 0; j < n; j++) {
79                     a[i][j] = mat[i][j];
80                 }
81             }
82
83             const double EPS = 1E-9;
84             T det = 1;
85
86             for (int i = 0; i < n; ++i) {
87                 int k = i;
88
89                 for (int j = i + 1; j < n; j++) {
90                     if (abs(a[j][i]) > abs(a[k][i])) {
91                         k = j;
92                     }
93                 }
94
95                 if (abs(a[k][i]) < EPS) {
96                     det = 0;
97                     break;
98                 }
99
100                 swap(a[i], a[k]);
101
102                 if (i != k) det = -det;
103
104                 det = det * a[i][i];
105
106                 for (int j = i + 1; j < n; j++) {
107                     a[i][j] = a[i][j] / a[i][i];
108                 }
109
110                 for (int j = 0; j < n; j++) {
111                     if (j != i && abs(a[j][i]) > EPS) {
112                         for (int k = i + 1; k < n; k++) {
113                             a[j][k] = a[j][k] - a[i][k] *
114                                 ↪ a[j][i];
115                         }
116                     }
117                 }
118             }
119         }
120     }

```

```

117     }
118
119     return det;
120 }
121 };

```

## 5 Strings

### 5.1 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 };

```

### 5.2 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
8  ↪ and
9  // its substring starting at i
10 vector<int> zFunction(const string& s) {
11     int n = s.length();
12     vector<int> z(n);
13     z[0] = n;
14     int l = 0;
15     int r = 0;
16
17     for (int i = 1; i < n; i++) {
18         if (i <= r) {
19             z[i] = min(z[i - l], r - i + 1);
20         }
21         while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
22             z[i]++;
23         }
24         if (i + z[i] - 1 > r) {
25             l = i;
26             r = i + z[i] - 1;
27         }
28     }
29     return z;
30 }

```

### 5.3 Suffix Array

```

1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  // sa[i] = the starting index of the ith suffix (starting
6  ↪ at 0)
7  // sorted in lexicographic order
8  vector<int> suffix_array(const string& s_, int alpha=256)
9  ↪ {
10     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
30            ↪ second to pair
31            if (pn[i] < 0) pn[i] += n; // cyclic
32        }
33
34        int num_groups = g[p[n - 1]] + 1;
35        fill(cnt.begin(), cnt.begin() + num_groups, 0);
36
37        // Radix sort
38        for (int i = 0; i < n; i++) cnt[g[pn[i]]]++;
39        for (int i = 1; i < num_groups; i++) cnt[i] +=
40            ↪ cnt[i - 1];
41        for (int i = n - 1; i >= 0; i--)
42            ↪ p[--cnt[g[pn[i]]]] = pn[i];
43        gn[p[0]] = 0;
44
45        for (int i = 1; i < n; i++) {
46            pair<int, int> prev, cur;
47            prev.first = g[p[i - 1]];
48            prev.second = g[p[i - 1]] + len - (p[i - 1] +
49            ↪ len >= n ? n : 0);
50            cur.first = g[p[i]];
51            cur.second = g[p[i]] + len - (p[i] + len >= n
52            ↪ ? n : 0);
53            gn[p[i]] = gn[p[i - 1]] + (cur != prev);

```

```
49     }
50     g.swap(gn);
51 }
52 p.erase(p.begin());
53 return p;
54 }
```

## 6 Flows

### 6.1 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 version should work on flow graph with float
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
28     // source
29     vector<int> ptr; // points to the next edge
30     // which can be used
31     vector<FlowEdge<T>> edges;
```

```
32     : n(_n), s(_s), t(_t), adj(_n), level(_n),
33     { ptr(_n) {}
34
35 void addEdge(int u, int v, int c, int rc=0) {
36     int eid = (int) edges.size();
37     adj[u].push_back(eid);
38     adj[v].push_back(eid + 1);
39     edges.emplace_back(u, v, c, 0);
40     edges.emplace_back(v, u, rc, 0);
41 }
42
43 bool bfs() {
44     fill(level.begin(), level.end(), -1);
45     level[s] = 0;
46     queue<int> q;
47     q.push(s);
48
49 while (!q.empty()) {
50     int u = q.front();
51     q.pop();
52
53 for (int eid : adj[u]) {
54     const auto& e = edges[eid];
55     if (e.c - e.f <= eps || level[e.v] != -1)
56         continue;
57     level[e.v] = level[u] + 1;
58     q.push(e.v);
59 }
60
61 return level[t] != -1;
62 }
63
64 T dfs(int u, T flow) {
65     if (u == t) return flow;
66
67 for (int& j = ptr[u]; j < (int) adj[u].size();
68     j++) {
69     int eid = adj[u][j];
70     const auto& e = edges[eid];
71     if (e.c - e.f > eps && level[e.v] == level[u]
72         + 1) {
73         T df = dfs(e.v, min(e.c - e.f, flow));
74         if (df > eps) {
75             edges[eid].f += df;
76             edges[eid ^ 1].f -= df;
77             return df;
78         }
79     }
80 }
```

```
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 };
```

## 7 Matching

### 7.1 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
10 // from 0
11 // Time complexity:  $O(\sqrt{|V|}|E|)$ 
12 struct HopcroftKarp {
13     const int INF = (int) 1e9;
14     int nu;
15     int nv;
16     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 &&
70              ↪ dfs(v_mate[v]))) {
71             v_mate[v] = u;
72             u_mate[u] = v;
73             return true;
74         }
75     }
76
77     return false;
78 }
79
80 vector<pair<int, int>> maxMatching() {
81     int matching = 0;
82
83     while (bfs()) { // there is at least 1 augmenting
84         ↪ path
85         for (int u = 0; u < nu; u++) {
86             if (u_mate[u] == -1 && dfs(u)) {
87                 ++matching;
88             }
89         }
90     }
91
92     vector<pair<int, int>> res;
93
94     for (int u = 0; u < nu; u++) {
95         if (u_mate[u] == -1) continue;
96         res.emplace_back(u, u_mate[u]);
97     }
98
99     assert(res.size() == matching);
100     return res;

```

## 8 Geometry

### 8.1 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 }

```

## 8.2 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 +
16     ↪ 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); } //
    ↪  $[-\pi, \pi]$ 
24 P unit() const { return *this / abs(); } // makes
    ↪ abs()=1
25 P perp() const { return P(-y, x); } // rotates  $+\pi/2$ 
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  $c \in \mathbb{R}$ )
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 };

```

### 8.3 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)) *
80             ↪ orient(r, p1, p2) > 0) {
81             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
107                 ↪ accepted, < otherwise
108                 if (orient(a, b, c) <= 0) break;
109                 hull.pop_back();
110             }
111             hull.push_back(c);
112         }
113         hull.pop_back();
114         reverse(points.begin(), points.end());
115     }
116
117     return Polygon<T>(hull);
118 }
```

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 ():

```
1 // by default: non-decreasing, v must be of
  ↪ comparator type
2 sort(v.begin(), v.end());
3 // custom comparator
4 sort(v.begin(), v.end(), [](const Obj& o1, const
  ↪ Obj& o2) {
5     return o1.x < o2.x;
6 });
```

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  $O(\text{length}(t))$
- 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*)