

Contents

1	Template	1
1.1	Makefile	1
1.2	vimrc	1
1.3	Stress testing	1
2	Graph	1
2.1	Dijkstra	1
2.2	Strongly Connected Components	2
2.3	Lowest Common Ancestor	3
2.4	Euler Path	3
3	Structures	4
3.1	Disjoint Set/Union-Find/Disjoint-Set-Union (DSU)	4
3.2	Segment Tree	4
3.3	Sparse Table	5
3.4	Sqrt Decomposition & Mo's algorithm	5
4	Maths	6
4.1	Modular Arithmetic	6
4.2	Modnum	7
4.3	Sieve of Eratosthenes	7
4.4	Primality Test	7
4.5	Euclidean Algorithm	7
4.6	Extended Euclidean Algorithm	8
4.7	Euler's Totient Function	8
4.8	Matrix	8
5	Strings	9
5.1	Trie	9
5.2	Z function	9
5.3	KMP	10
5.4	Suffix Array	10
6	Flows	10
6.1	Dinic Max Flow	10
7	Matching	11
7.1	Hopcroft-Karp Bipartite Matching	11
8	Geometry	12
8.1	Utility	12
8.2	Point	12
8.3	Polygon	13
9	C++ STL	14
9.1	vector	14
9.2	set	14
9.3	map	14
9.4	unordered_set and unordered_map	14
9.5	pair	14
9.6	string	14
9.7	Other useful utilities	15

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
```

1.3 Stress testing

```
#!/bin/bash

for((i = 1; ; ++i)); do
    echo $i
    python3 gen.py $i > inp.txt
    ./main < inp.txt > out.txt
    ./slow < inp.txt > ans.txt
    diff -w out.txt ans.txt || break
done
```

2 Graph

2.1 Dijkstra

```
#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]);
                }
            }
        }
    }
};
```

```

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 int main() {
75     int n, m, s, t;
76     cin >> n >> m >> s >> t;
77
78     Graph g(n);
79
80     for (int i = 0; i < m; i++) {
81         int u, v, w;
82         cin >> u >> v >> w;
83         g.addEdge(u, v, w);
84     }
85
86     int64_t dist = g.dijkstra(s, t);
87
88     if (dist != g.inf) {
89         vector<Edge> path = g.getShortestPath(s, t);
90         cout << dist << ' ' << path.size() << '\n';
91         for (Edge e : path) cout << e.u << ' ' << e.v <<
92             < '\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
10     ↪ topological order
11 struct Graph {
12     int n;
13     vector<vector<int>> adj;
14     vector<vector<int>> adj_t;
15     vector<int> mark;
16     vector<int> order;
17     vector<int> leader;
18     vector<vector<int>> components;
19
20     Graph(int n_) : n(n_), adj(n), adj_t(n),
21         mark(n), leader(n) {}
22
23     void addEdge(int u, int v) {
24         adj[u].push_back(v);
25         adj_t[v].push_back(u);
26     }
27
28     void dfsForward(int u) {
29         assert(mark[u] == 0);
30         mark[u] = 1;
31
32         for (int v : adj[u]) {
33             if (mark[v] == 0) {
34                 dfsForward(v);
35             }
36         }
37
38         order.push_back(u);
39     }
40
41     void dfsBackward(int u, int p) {
42         assert(mark[u] == 1);
43         mark[u] = 2;
44         leader[u] = p;

```

```

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 }

```

2.3 Lowest Common Ancestor

```

1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  // https://judge.yosupo.jp/problem/lca
6  struct LCA {
7      vector<int> height, euler, first, segtree;
8      vector<bool> visited;
9      int n;
10
11      LCA(vector<vector<int>> &adj, int root = 0) {
12          n = adj.size();
13          height.resize(n);
14          first.resize(n);
15          euler.reserve(n * 2);
16          visited.assign(n, false);
17          dfs(adj, root);
18          int m = euler.size();
19          segtree.resize(m * 4);
20          build(1, 0, m - 1);
21      }
22
23      void dfs(vector<vector<int>> &adj, int node, int h =
          ↳ 0) {
24          visited[node] = true;
25          height[node] = h;
26          first[node] = euler.size();
27          euler.push_back(node);
28          for (auto to : adj[node]) {
29              if (!visited[to]) {
30                  dfs(adj, to, h + 1);
31                  euler.push_back(node);

```

```

32          }
33      }
34  }
35
36  void build(int node, int b, int e) {
37      if (b == e) {
38          segtree[node] = euler[b];
39      } else {
40          int mid = (b + e) / 2;
41          build(node << 1, b, mid);
42          build(node << 1 | 1, mid + 1, e);
43          int l = segtree[node << 1], r = segtree[node
          ↳ << 1 | 1];
44          segtree[node] = (height[l] < height[r]) ? l :
          ↳ r;
45      }
46  }
47
48  int query(int node, int b, int e, int L, int R) {
49      if (b > R || e < L)
50          return -1;
51      if (b >= L && e <= R)
52          return segtree[node];
53      int mid = (b + e) >> 1;
54
55      int left = query(node << 1, b, mid, L, R);
56      int right = query(node << 1 | 1, mid + 1, e, L,
          ↳ R);
57      if (left == -1) return right;
58      if (right == -1) return left;
59      return height[left] < height[right] ? left :
          ↳ right;
60  }
61
62  int lca(int u, int v) {
63      int left = first[u], right = first[v];
64      if (left > right)
65          swap(left, right);
66      return query(1, 0, euler.size() - 1, left,
          ↳ right);
67  }
68  };
69
70  int main() {
71      int n, q;
72      cin >> n >> q;
73

```

```

74     vector<vector<int>> adj(n);
75
76     for (int u = 1; u < n; u++) {
77         int v;
78         cin >> v;
79         adj[u].push_back(v);
80         adj[v].push_back(u);
81     }
82
83     LCA solver(adj);
84
85     for (int i = 0; i < q; i++) {
86         int u, v;
87         cin >> u >> v;
88         cout << solver.lca(u, v) << '\n';
89     }
90
91     return 0;
92 }

```

2.4 Euler Path

```

1  /// Hierholzer's Algorithm for Euler path (uses every
          ↳ edge exactly once)
2
3  #pragma once
4
5  #include <bits/stdc++.h>
6
7  using namespace std;
8
9  vector<int> findDirectedEulerPath(vector<vector<int>>&
          ↳ adj, int s) {
10      int n = adj.size();
11
12      vector<int> edge_id(n, 0);
13      stack<int> st;
14      vector<int> res;
15
16      st.push(s);
17
18      while (!st.empty()) {
19          int u = st.top();
20          st.pop();
21

```

```

22     while (edge_id[u] < (int) adj[u].size()) {
23         st.push(u);
24         u = adj[u][edge_id[u]++];
25     }
26
27     res.push_back(u);
28 }
29
30 reverse(res.begin(), res.end());
31 return res;
32 }

```

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, 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; }

```

```

37 void build(vector<int>& a, int u, int from, int to) {
38     if (from == to) {
39         t[u] = Node({from, to, a[from], dlazy,
40             ↪ false});
41         return;
42     }
43
44     int l = left(u);
45     int r = right(u);
46     int mid = (from + to) / 2;
47     build(a, l, from, mid);
48     build(a, r, mid + 1, to);
49     T val = merge(t[l].val, t[r].val);
50     t[u] = Node({from, to, val, dlazy, false});
51 }
52
53 T query(int from, int to, int u=0) {
54     if (from <= t[u].from && t[u].to <= to) return
55         ↪ t[u].val;
56     if (to < t[u].from || t[u].to < from) return
57         ↪ dquery;
58     pushDown(u);
59     return merge(query(from, to, left(u)),
60         ↪ query(from, to, right(u)));
61 }
62
63 void update(int from, int to, T delta, int u=0) {
64     if (from > to) return;
65
66     if (from == t[u].from && to == t[u].to) {
67         apply(u, delta);
68         return;
69     }
70
71     pushDown(u);
72     int l = left(u);
73     int r = right(u);
74     int mid = (t[u].from + t[u].to) / 2;
75     update(from, min(to, mid), delta, l);
76     update(max(from, mid + 1), to, delta, r);
77     t[u].val = merge(t[l].val, t[r].val);
78 }
79
80 template <typename T>
81 struct SegmentAssignUpdate : public SegmentTree<T> {

```

```

80 SegmentAssignUpdate(vector<int>& a, BinOp<T> merge_,
   ↳ int dquery_) :
81     SegmentTree<T>(a, 0, merge_, dquery_) {}
82
83 virtual void apply(int u, T delta) {
84     auto& t = this->t;
85     t[u].val = delta;
86     t[u].is_lazy = true;
87 }
88
89 virtual void pushDown(int u) {
90     auto& t = this->t;
91     int l = this->left(u);
92     int r = this->right(u);
93     if (t[u].is_lazy) {
94         t[l].val = t[r].val = t[u].val;
95         t[l].is_lazy = t[r].is_lazy = true;
96         t[u].is_lazy = false;
97     }
98 }
99 };
100
101 template <typename T>
102 struct SegmentAddUpdate : SegmentTree<T> {
103     SegmentAddUpdate(vector<int>& a, int dlazy_, BinOp<T>
   ↳ merge_, int dquery_) :
104         SegmentTree<T>(a, dlazy_, merge_, dquery_) {}
105
106 virtual void apply(int u, T delta) {
107     auto& t = this->t;
108     t[u].val += delta;
109     t[u].lazy += delta;
110 }
111
112 virtual void pushDown(int u) {
113     auto& t = this->t;
114     int l = this->left(u);
115     int r = this->right(u);
116     t[l].val += t[u].lazy;
117     t[l].lazy += t[u].lazy;
118     t[r].val += t[u].lazy;
119     t[r].lazy += t[u].lazy;
120     t[u].lazy = 0;
121 }
122 };

```

3.3 Sparse Table

```

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 // Queries on immutable array
11 template <typename T>
12 class SparseTable {
13 public:
14     int n;
15     vector<vector<T>> mat;
16     BinOp<T> f;
17
18     SparseTable(const vector<T>& a, const BinOp<T>& f_)
   ↳ : f(f_) {
19         n = static_cast<int>(a.size());
20         int max_log = 32 - __builtin_clz(n);
21         mat.resize(max_log);
22         mat[0] = a;
23         for (int j = 1; j < max_log; j++) {
24             mat[j].resize(n - (1 << j) + 1);
25             for (int i = 0; i <= n - (1 << j); i++) {
26                 mat[j][i] = f(mat[j - 1][i], mat[j - 1][i
   ↳ + (1 << (j - 1))]);
27             }
28         }
29     }
30
31     T get(int from, int to) const {
32         assert(0 <= from && from <= to && to <= n - 1);
33         int lg = 32 - __builtin_clz(to - from + 1) - 1;
34         return f(mat[lg][from], mat[lg][to - (1 << lg) +
   ↳ 1]);
35     }
36 };

```

3.4 Sqrt Decomposition & Mo's algorithm

```

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

```

```

3  using namespace std;
4
5  struct Query {
6      int from, to, index;
7      Query(int from, int to, int index) :
8          from(from), to(to), index(index) {}
9  };
10
11 struct MoSolver {
12     vector<int> a;
13     vector<Query> queries;
14     int n;
15     int k;
16     int total;
17     unordered_map<int, int> freq;
18
19     MoSolver(vector<int>& a, vector<Query>& queries, int
   ↳ n, int k)
20         : a(a), queries(queries), n(n), k(k), total(0) {}
21
22     void add(int index) {
23         int x = a[index];
24         if (freq.find(x) == freq.end()) freq[x] = 0;
25         int y = k - x;
26
27         if (freq.find(y) != freq.end()) {
28             if (y != x) {
29                 if (freq[y] > freq[x]) {
30                     total++;
31                 }
32             } else {
33                 if (freq[x] % 2 == 1) {
34                     total++;
35                 }
36             }
37         }
38
39         freq[x]++;
40     }
41
42     void remove(int index) {
43         int x = a[index];
44         int y = k - x;
45
46         if (freq.find(y) != freq.end()) {
47             if (y != x) {

```

```

48         if (freq[y] >= freq[x]) {
49             total--;
50         }
51     } else {
52         if (freq[x] % 2 == 0) {
53             total--;
54         }
55     }
56 }
57
58 freq[x]--;
59 }
60
61 vector<int> solve() {
62     const int blockSize = sqrt(n);
63     sort(queries.begin(), queries.end(),
64          [&](const Query& q1, const Query& q2) {
65             if (q1.from / blockSize != q2.from /
66                 ↳ blockSize) {
67                 return q1.from / blockSize < q2.from
68                     ↳ / blockSize;
69             }
70             return q1.to < q2.to;
71         });
72
73     vector<int> ans(queries.size());
74
75     int from = 0, to = -1;
76
77     for (Query& q : queries) {
78         while (from > q.from) {
79             from--;
80             add(from);
81         }
82         while (to < q.to) {
83             to++;
84             add(to);
85         }
86         while (from < q.from) {
87             remove(from);
88             from++;
89         }
90         while (to > q.to) {
91             remove(to);
92             to--;
93         }
94     }

```

```

93         ans[q.index] = total;
94     }
95
96     return ans;
97 }
98 };
99
100 struct Solver {
101     Solver(int n, int m, int k) {
102         vector<int> a(n);
103
104         for (int i = 0; i < n; i++) {
105             cin >> a[i];
106         }
107
108         vector<Query> queries;
109
110         for (int i = 0; i < m; i++) {
111             int from, to;
112             cin >> from >> to;
113             from--; to--;
114             if (from > to) swap(from, to);
115             queries.emplace_back(from, to, i);
116         }
117
118         MoSolver moSolver(a, queries, n, k);
119         vector<int> ans = moSolver.solve();
120
121         for (int i = 0; i < (int) ans.size(); i++) {
122             cout << ans[i] << '\n';
123         }
124     }
125 };
126
127 int main() {
128     ios_base::sync_with_stdio(false);
129     cin.tie(nullptr);
130
131     while (true) {
132         int n, m, k;
133         cin >> n >> m >> k;
134         if (n == 0 && m == 0 && k == 0) break;
135         Solver(n, m, k);
136         cout << "\n";
137     }
138
139     return 0;

```

```

140 }

```

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 }
17
18 int modSub(int a, int b, int mod) {
19     a -= b;
20     if (a < 0) a += mod;
21     return a;
22 }
23
24 int modMul(int a, int b, int mod) {
25     int64_t res = (int64_t) a * b;
26     return (int) (res % mod);
27 }
28
29 int64_t binPow(int64_t a, int64_t x) {
30     int64_t res = 1;
31     while (x) {
32         if (x & 1) res *= a;
33         a *= a;
34         x >>= 1;
35     }
36     return res;
37 }
38
39 int64_t modPow(int64_t a, int64_t x, int mod) {
40     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;

```

```

36     }
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 }

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

```

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     Matrix operator+(const Matrix& o) const {
20         assert(n == o.n);
21         const Matrix& a = *this;
22         Matrix res(n);
23
24         for (int i = 0; i < n; i++) {
25             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     }
60     return res;
61 }
62
63 void identity() {
64     Matrix& a = *this;
65     for (int i = 0; i < n; i++) {
66         for (int j = 0; j < n; j++) {
67             if (i == j) a[i][j] = 1;
68             else a[i][j] = 0;
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            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     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 }

```

```

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 }

```

5.3 KMP

```

1 #pragma once
2
3 #include <bits/stdc++.h>
4
5 using namespace std;
6
7 // f[i]: length of the longest proper prefix of
8 // the substring s[0..i] which is also a suffix of
9 // this substring
10 vector<int> kmp(string& s) {
11     int n = (int)s.length();
12     vector<int> f(n);
13     for (int i = 1; i < n; i++) {
14         int j = f[i - 1];
15         while (j > 0 && s[i] != s[j]) {
16             j = f[j - 1];
17         }
18         if (s[i] == s[j]) {
19             j++;
20         }
21         f[i] = j;
22     }
23     return f;
24 }

```

5.4 Suffix Array

```

1 #include <bits/stdc++.h>
2
3 using namespace std;
4

```

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 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)
55                 ↪ continue;
56             level[e.v] = level[u] + 1;
57             q.push(e.v);
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();
67             ↪ j++) {
68             int eid = adj[u][j];
69             const auto& e = edges[eid];
70             if (e.c - e.f > eps && level[e.v] == level[u]
71                 ↪ + 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     return f;
97 }
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
   // 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 &&
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;
101 }

```

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); }
17     P operator-(const P &o) const { return P(x - o.x, y -
18         ↪ o.y); }
19     P operator*(T d) const { return P(x * d, y * d); }
20     P operator/(T d) const { return P(x / d, y / d); }
21     T dot(P o) const { return x * o.x + y * o.y; }
22     T cross(P o) const { return x * o.y - y * o.x; }
23     T abs2() const { return x * x + y * y; }
24     long double abs() const { return sqrt((long double)
25         ↪ abs2()); }
26     double angle() const { return atan2(y, x); } //
27     ↪ [-π, π]
28     P unit() const { return *this / abs(); } // makes
29     ↪ abs()=1
30     P perp() const { return P(-y, x); } // rotates +π/2
31
32     P rotate(double a) const { // ccw
33         return P(x * cos(a) - y * sin(a), x * sin(a) + y
34             ↪ * cos(a));
35     }
36
37     friend istream &operator>>(istream &is, P &p) {
38         return is >> p.x >> p.y;
39     }
40
41     friend ostream &operator<<(ostream &os, P &p) {
42         return os << "(" << p.x << ", " << p.y << ")";
43     }
44
45     // position of c relative to a->b
46     // > 0: c is on the left of a->b
47     friend T orient(P a, P b, P c) {
48         return (b - a).cross(c - a);
49     }
50
51     // Check if  $\vec{u}$  and  $\vec{v}$  are parallel
52     // ( $\vec{u} = c\vec{v}$ ) where  $c \in R$ 
53     friend bool parallel(P u, P v) {
54         return u.cross(v) == 0;
55     }
56
57     // Check if point p lies on the segment ab
58     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
62         ↪ true;
63     if (max(p1.y, q1.y) < min(p2.y, q2.y)) return
64         ↪ true;
65     if (max(p2.x, q2.x) < min(p1.x, q1.x)) return
66         ↪ true;
67     if (max(p2.y, q2.y) < min(p1.y, q1.y)) return
68         ↪ true;
69     return false;
70 }
71
72 friend bool intersect(P p1, P p2, P p3, P p4) {
73     // Check if two segments are parallel
74     if (parallel(p2 - p1, p4 - p3)) {
75         // Check if 4 ps are colinear
76         if (!parallel(p2 - p1, p3 - p1)) return
77             ↪ false;
78         if (boundingBox(p1, p2, p3, p4)) return
79             ↪ false;
80         return true;
81     }
82     // check if one line is completely on one side of
83     ↪ the other
84     for (int i = 0; i < 2; i++) {
85         if (sgn(orient(p1, p2, p3)) == sgn(orient(p1,
86             ↪ p2, p4))
87             && sgn(orient(p1, p2, p3)) != 0) {
88             return false;
89         }
90         swap(p1, p3);
91         swap(p2, p4);
92     }
93     return true;
94 }
95
96 // Check if p is in ∠bac (including the rays)
97 friend bool inAngle(P a, P b, P c, P p) {
98     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) <=
94         ↪ 0;
95 }
96 // 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)) *
    ↪ 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
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 $O(length(t))$
- 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)`