

# Team Reference Document

Heltion

January 10, 2024

Contents

<b>1</b>	<b>Contest</b>	<b>1</b>
1.1	Makefile . . . . .	1
1.2	.clang-format . . . . .	1
1.3	debug.h . . . . .	1
1.4	Template . . . . .	1
1.5	pbds . . . . .	1
<b>2</b>	<b>Graph</b>	<b>1</b>
2.1	Connected Components . . . . .	1
2.1.1	Strongly Connected Components . . . . .	1
2.1.2	Two-vertex-connected Components . . . . .	2
2.1.3	Two-edge-connected Components . . . . .	2
2.1.4	Three-edge-connected Components . . . . .	2
2.2	Euler Walks . . . . .	3
2.3	Dominator Tree . . . . .	4
2.4	Directed Minimum Spanning Tree . . . . .	4
2.5	K Shortest Paths . . . . .	5
2.6	Global Minimum Cut . . . . .	5
2.7	Minimum Perfect Matching on Bipartite Graph . . . . .	6
2.8	Matching on General Graph . . . . .	6
2.9	Maximum Flow . . . . .	7
2.10	Minimum Cost Maximum Flow . . . . .	8
<b>3</b>	<b>Data Structure</b>	<b>9</b>
3.1	Disjoint Set Union . . . . .	9
3.2	Sparse Table . . . . .	9
3.3	Treap . . . . .	10
3.4	Lines Maximum . . . . .	10
3.5	Segments Maximum . . . . .	11
3.6	Segment Beats . . . . .	11
3.7	Tree . . . . .	12
3.7.1	Least Common Ancestor . . . . .	12
3.7.2	Link Cut Tree . . . . .	12
<b>4</b>	<b>String</b>	<b>13</b>
4.1	Z . . . . .	13
4.2	Lyndon Factorization . . . . .	14
4.3	Border . . . . .	14
4.4	Manacher . . . . .	14
4.5	Suffix Array . . . . .	14
4.6	Aho-Corasick Automaton . . . . .	14
4.7	Suffix Automaton . . . . .	15
4.8	Palindromic Tree . . . . .	15
<b>5</b>	<b>Number Theory</b>	<b>16</b>
5.1	Gaussian Integer . . . . .	16
5.2	Modular Arithmetic . . . . .	16
5.2.1	Sqrt . . . . .	16
5.2.2	Logarithm . . . . .	16
5.3	Chinese Remainder Theorem . . . . .	16
5.4	Miller Rabin . . . . .	17
5.5	Pollard Rho . . . . .	17
5.6	Primitive Root . . . . .	17
5.7	Sum of Floor . . . . .	17
5.8	Minimum of Remainder . . . . .	18
5.9	Stern Brocot Tree . . . . .	18
5.10	Nim Product . . . . .	18
<b>6</b>	<b>Numerical</b>	<b>19</b>
6.1	Golden Search . . . . .	19
6.2	Adaptive Simpson . . . . .	19
6.3	Simplex . . . . .	19
6.4	Green’s Theorem . . . . .	20
6.5	Double Integral . . . . .	20
<b>7</b>	<b>Convolution</b>	<b>20</b>
7.1	Fast Fourier Transform on $\mathbb{C}$ . . . . .	20
7.2	Formal Power Series on $\mathbb{F}_p$ . . . . .	20
7.2.1	Newton’s Method . . . . .	20
7.2.2	Arithmetic . . . . .	20
7.2.3	Interpolation . . . . .	20
7.2.4	Primes with root 3 . . . . .	21
7.3	Circular Transform . . . . .	21
7.4	Truncated Transform . . . . .	21
<b>8</b>	<b>Geometry</b>	<b>21</b>
8.1	Pick’s Theorem . . . . .	21
8.2	2D Geometry . . . . .	21

# 1 Contest

## 1.1 Makefile

```
1 %:%.cpp
2     g++ $< -o $@ -std=gnu++20 -O2 -Wall -Wextra \
3     -D_GLIBCXX_DEBUG -D_GLIBCXX_DEBUG_PEDANTIC
```

## 1.2 .clang-format

```
1 BasedOnStyle: Chromium
2 IndentWidth: 2
3 TabWidth: 2
4 AllowShortIfStatementsOnASingleLine: true
5 AllowShortLoopsOnASingleLine: true
6 AllowShortBlocksOnASingleLine: true
7 AllowShortFunctionsOnASingleLine: All
8 AlwaysBreakTemplateDeclarations: false
9 ColumnLimit: 77
```

## 1.3 debug.h

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 template <class T, size_t size = tuple_size<T>::value>
4 string to_debug(T, string s = "")
5     requires(not ranges::range<T>);
6 string to_debug(auto x)
7     requires requires(ostream& os) { os << x; }
8 {
9     return static_cast<ostringstream>(ostringstream() << x).str();
10 }
11 string to_debug(ranges::range auto x, string s = "")
12     requires(not is_same_v<decltype(x), string>)
13 {
14     for (auto xi : x) { s += ", " + to_debug(xi); }
15     return "[" + s.substr(s.empty() ? 0 : 2) + "]";
16 }
17 template <class T, size_t size>
18 string to_debug(T x, string s)
19     requires(not ranges::range<T>)
20 {
21     [&]<size_t... I>(index_sequence<I...>) {
22         ((s += ", " + to_debug(get<I>(x))), ...);
23     }(make_index_sequence<size>());
24     return "(" + s.substr(s.empty() ? 0 : 2) + ")";
25 }
26 #define debug(...) \
27     cerr << __FILE__ ":" << __LINE__ \
28     << " : " #__VA_ARGS__ " " << to_debug(tuple(__VA_ARGS__)) << "\n"
```

## 1.4 Template

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 using i64 = int64_t;
4 #ifndef ONLINE_JUDGE
5 #include "debug.h"
6 #else
7 #define debug(...) 417
8 #endif
9 int main() {
10     cin.tie(nullptr)->sync_with_stdio(false);
11     cout << fixed << setprecision(20);
12 }
```

## 1.5 pbds

```
1 #include <bits/extc++.h>
2 using namespace std;
3 using namespace __gnu_cxx;
4 using namespace __gnu_pbds;
5 using t = tree<int,
6             null_type,
7             less<int>,
8             rb_tree_tag,
9             tree_order_statistics_node_update>;
10 using p = __gnu_pbds::priority_queue<int, less<int>, pairing_heap_tag>;
```

# 2 Graph

## 2.1 Connected Components

### 2.1.1 Strongly Connected Components

Returns strongly connected components in topologically order.

```
1 vector<vector<int>>>
2 strongly_connected_components(const vector<vector<int>>> &g) {
3     int n = g.size();
4     vector<bool> done(n);
5     vector<int> pos(n, -1), stack;
6     vector<vector<int>>> res;
7     function<int(int)> dfs = [&](int u) {
8         int low = pos[u] = stack.size();
9         stack.push_back(u);
10        for (int v : g[u]) {
11            if (not done[v]) { low = min(low, ~pos[v] ? pos[v] : dfs(v)); }
12        }
13        if (low == pos[u]) {
14            res.emplace_back(stack.begin() + low, stack.end());
```

```

15     for (int v : res.back()) { done[v] = true; }
16     stack.resize(low);
17 }
18 return low;
19 };
20 for (int i = 0; i < n; i += 1) {
21     if (not done[i]) { dfs(i); }
22 }
23 ranges::reverse(res);
24 return res;
25 }

```

## 2.1.2 Two-vertex-connected Components

```

1 vector<vector<int>>
2 two_vertex_connected_components(const vector<vector<int>> &g) {
3     int n = g.size();
4     vector<int> pos(n, -1), stack;
5     vector<vector<int>> res;
6     function<int(int, int)> dfs = [&](int u, int p) {
7         int low = pos[u] = stack.size(), son = 0;
8         stack.push_back(u);
9         for (int v : g[u]) {
10             if (v != p) {
11                 if (~pos[v]) {
12                     low = min(low, pos[v]);
13                 } else {
14                     int end = stack.size(), lowv = dfs(v, u);
15                     low = min(low, lowv);
16                     if (lowv >= pos[u] and (~p or son++)) {
17                         res.emplace_back(stack.begin() + end, stack.end());
18                         res.back().push_back(u);
19                         stack.resize(end);
20                     }
21                 }
22             }
23         }
24         return low;
25 };
26 for (int i = 0; i < n; i += 1) {
27     if (pos[i] == -1) {
28         dfs(i, -1);
29         res.emplace_back(move(stack));
30     }
31 }
32 return res;
33 }

```

## 2.1.3 Two-edge-connected Components

```

1 vector<vector<int>> bcc(const vector<vector<int>> &g) {
2     int n = g.size();
3     vector<int> pos(n, -1), stack;
4     vector<vector<int>> res;
5     function<int(int, int)> dfs = [&](int u, int p) {
6         int low = pos[u] = stack.size(), pc = 0;
7         stack.push_back(u);
8         for (int v : g[u]) {
9             if (~pos[v]) {
10                 if (v != p or pc++) { low = min(low, pos[v]); }
11             } else {
12                 low = min(low, dfs(v, u));
13             }
14         }
15         if (low == pos[u]) {
16             res.emplace_back(stack.begin() + low, stack.end());
17             stack.resize(low);
18         }
19         return low;
20 };
21 for (int i = 0; i < n; i += 1) {
22     if (pos[i] == -1) { dfs(i, -1); }
23 }
24 return res;
25 }

```

## 2.1.4 Three-edge-connected Components

```

1 vector<vector<int>>
2 three_edge_connected_components(const vector<vector<int>> &g) {
3     int n = g.size(), dft = -1;
4     vector<int> pre(n, -1), post(n), path(n, -1), low(n), deg(n);
5     DisjointSetUnion dsu(n);
6     function<void(int, int)> dfs = [&](int u, int p) {
7         int pc = 0;
8         low[u] = pre[u] = dft += 1;
9         for (int v : g[u]) {
10             if (v != u and (v != p or pc++)) {
11                 if (pre[v] != -1) {
12                     if (pre[v] < pre[u]) {
13                         deg[u] += 1;
14                         low[u] = min(low[u], pre[v]);
15                     } else {
16                         deg[u] -= 1;
17                         for (int &p = path[u];
18                             p != -1 and pre[p] <= pre[v] and pre[v] <= post[p];) {
19                             dsu.merge(u, p);
20                             deg[u] += deg[p];
21                             p = path[p];
22                         }
23                     }
24                 }
25             }
26         }
27     };
28     dfs(0, -1);
29     for (int i = 0; i < n; i += 1) {
30         if (pre[i] == -1) { dfs(i, -1); }
31     }
32     return res;
33 }

```

```

24     } else {
25         dfs(v, u);
26         if (path[v] == -1 and deg[v] <= 1) {
27             low[u] = min(low[u], low[v]);
28             deg[u] += deg[v];
29         } else {
30             if (deg[v] == 0) { v = path[v]; }
31             if (low[u] > low[v]) {
32                 low[u] = min(low[u], low[v]);
33                 swap(v, path[u]);
34             }
35             for (; v != -1; v = path[v]) {
36                 dsu.merge(u, v);
37                 deg[u] += deg[v];
38             }
39         }
40     }
41 }
42 }
43 post[u] = dft;
44 };
45 for (int i = 0; i < n; i += 1) {
46     if (pre[i] == -1) { dfs(i, -1); }
47 }
48 vector<vector<int>> _res(n);
49 for (int i = 0; i < n; i += 1) { _res[dsu.find(i)].push_back(i); }
50 vector<vector<int>> res;
51 for (auto &res_i : _res) {
52     if (not res_i.empty()) { res.emplace_back(move(res_i)); }
53 }
54 return res;
55 }

```

## 2.2 Euler Walks

```

1 optional<vector<vector<pair<int, bool>>>>
2 undirected_walks(int n, const vector<pair<int, int>> &edges) {
3     int m = ssize(edges);
4     vector<vector<pair<int, bool>>> res;
5     if (not m) { return res; }
6     vector<vector<pair<int, bool>>> g(n);
7     for (int i = 0; i < m; i += 1) {
8         auto [u, v] = edges[i];
9         g[u].emplace_back(i, true);
10        g[v].emplace_back(i, false);
11    }
12    for (int i = 0; i < n; i += 1) {
13        if (g[i].size() % 2) { return {}; }
14    }
15    vector<pair<int, bool>> walk;
16    vector<bool> visited(m);
17    vector<int> cur(n);

```

```

18    function<void(int)> dfs = [&](int u) {
19        for (int &i = cur[u]; i < ssize(g[u]);) {
20            auto [j, d] = g[u][i];
21            if (not visited[j]) {
22                visited[j] = true;
23                dfs(d ? edges[j].second : edges[j].first);
24                walk.emplace_back(j, d);
25            } else {
26                i += 1;
27            }
28        }
29    };
30    for (int i = 0; i < n; i += 1) {
31        dfs(i);
32        if (not walk.empty()) {
33            ranges::reverse(walk);
34            res.emplace_back(move(walk));
35        }
36    }
37    return res;
38 }
39 optional<vector<vector<int>>>
40 directed_walks(int n, const vector<pair<int, int>> &edges) {
41     int m = ssize(edges);
42     vector<vector<int>> res;
43     if (not m) { return res; }
44     vector<int> d(n);
45     vector<vector<int>> g(n);
46     for (int i = 0; i < m; i += 1) {
47         auto [u, v] = edges[i];
48         g[u].push_back(i);
49         d[v] += 1;
50     }
51     for (int i = 0; i < n; i += 1) {
52         if (ssize(g[i]) != d[i]) { return {}; }
53     }
54     vector<int> walk;
55     vector<int> cur(n);
56     vector<bool> visited(m);
57     function<void(int)> dfs = [&](int u) {
58         for (int &i = cur[u]; i < ssize(g[u]);) {
59             int j = g[u][i];
60             if (not visited[j]) {
61                 visited[j] = true;
62                 dfs(edges[j].second);
63                 walk.push_back(j);
64             } else {
65                 i += 1;
66             }
67         }
68     };
69     for (int i = 0; i < n; i += 1) {
70         dfs(i);

```

```

71 if (not walk.empty()) {
72     ranges::reverse(walk);
73     res.emplace_back(move(walk));
74 }
75 }
76 return res;
77 }

```

## 2.3 Dominator Tree

```

1 vector<int> dominator(const vector<vector<int>>& g, int s) {
2     int n = g.size();
3     vector<int> pos(n, -1), p, label(n), dom(n), sdом(n), dsu(n), par(n);
4     vector<vector<int>> rg(n), bucket(n);
5     function<void(int)> dfs = [&](int u) {
6         int t = p.size();
7         p.push_back(u);
8         label[t] = sdом[t] = dsu[t] = pos[u] = t;
9         for (int v : g[u]) {
10             if (pos[v] == -1) {
11                 dfs(v);
12                 par[pos[v]] = t;
13             }
14             rg[pos[v]].push_back(t);
15         }
16     };
17     function<int(int, int)> find = [&](int u, int x) {
18         if (u == dsu[u]) { return x ? -1 : u; }
19         int v = find(dsu[u], x + 1);
20         if (v < 0) { return u; }
21         if (sdом[label[dsu[u]]] < sdом[label[u]]) { label[u] = label[dsu[u]]; }
22         dsu[u] = v;
23         return x ? v : label[u];
24     };
25     dfs(s);
26     iota(dom.begin(), dom.end(), 0);
27     for (int i = ssize(p) - 1; i >= 0; i -= 1) {
28         for (int j : rg[i]) { sdом[i] = min(sdом[i], sdом[find(j, 0)]); }
29         if (i) { bucket[sdом[i]].push_back(i); }
30         for (int k : bucket[i]) {
31             int j = find(k, 0);
32             dom[k] = sdом[j] == sdом[k] ? sdом[j] : j;
33         }
34         if (i > 1) { dsu[i] = par[i]; }
35     }
36     for (int i = 1; i < ssize(p); i += 1) {
37         if (dom[i] != sdом[i]) { dom[i] = dom[dom[i]]; }
38     }
39     vector<int> res(n, -1);
40     res[s] = s;
41     for (int i = 1; i < ssize(p); i += 1) { res[p[i]] = p[dom[i]]; }
42     return res;

```

```

43 }

```

## 2.4 Directed Minimum Spanning Tree

```

1 struct Node {
2     Edge e;
3     int d;
4     Node *l, *r;
5     Node(Edge e) : e(e), d(0) { l = r = nullptr; }
6     void add(int v) {
7         e.w += v;
8         d += v;
9     }
10    void push() {
11        if (l) { l->add(d); }
12        if (r) { r->add(d); }
13        d = 0;
14    }
15 };
16 Node *merge(Node *u, Node *v) {
17     if (not u or not v) { return u ? v : u; }
18     if (u->e.w > v->e.w) { swap(u, v); }
19     u->push();
20     u->r = merge(u->r, v);
21     swap(u->l, u->r);
22     return u;
23 }
24 void pop(Node *&u) {
25     u->push();
26     u = merge(u->l, u->r);
27 }
28 pair<i64, vector<int>>
29 directed_minimum_spanning_tree(int n, const vector<Edge> &edges, int s) {
30     i64 ans = 0;
31     vector<Node *> heap(n), edge(n);
32     RollbackDisjointSetUnion dsu(n), rbdсу(n);
33     vector<pair<Node *, int>> cycles;
34     for (auto e : edges) { heap[e.v] = merge(heap[e.v], new Node(e)); }
35     for (int i = 0; i < n; i += 1) {
36         if (i == s) { continue; }
37         for (int u = i;;) {
38             if (not heap[u]) { return {}; }
39             ans += (edge[u] = heap[u])->e.w;
40             edge[u]->add(-edge[u]->e.w);
41             int v = rbdсу.find(edge[u]->e.u);
42             if (dsu.merge(u, v)) { break; }
43             int t = rbdсу.time();
44             while (rbdсу.merge(u, v)) {
45                 heap[rbdсу.find(u)] = merge(heap[u], heap[v]);
46                 u = rbdсу.find(u);
47                 v = rbdсу.find(edge[v]->e.u);
48             }

```

```

49     cycles.emplace_back(edge[u], t);
50     while (heap[u] and rbdsu.find(heap[u]->e.u) == rbdsu.find(u)) {
51         pop(heap[u]);
52     }
53 }
54 }
55 for (auto [p, t] : cycles | views::reverse) {
56     int u = rbdsu.find(p->e.v);
57     rbdsu.rollback(t);
58     int v = rbdsu.find(edge[u]->e.v);
59     edge[v] = exchange(edge[u], p);
60 }
61 vector<int> res(n, -1);
62 for (int i = 0; i < n; i += 1) { res[i] = i == s ? i : edge[i]->e.u; }
63 return {ans, res};
64 }

```

## 2.5 K Shortest Paths

```

1 struct Node {
2     int v, h;
3     i64 w;
4     Node *l, *r;
5     Node(int v, i64 w) : v(v), w(w), h(1) { l = r = nullptr; }
6 };
7 Node *merge(Node *u, Node *v) {
8     if (not u or not v) { return u ?: v; }
9     if (u->w > v->w) { swap(u, v); }
10    Node *p = new Node(*u);
11    p->r = merge(u->r, v);
12    if (p->r and (not p->l or p->l->h < p->r->h)) { swap(p->l, p->r); }
13    p->h = (p->r ? p->r->h : 0) + 1;
14    return p;
15 }
16 struct Edge {
17     int u, v, w;
18 };
19 template <typename T>
20 using minimum_heap = priority_queue<T, vector<T>, greater<T>>;
21 vector<i64> k_shortest_paths(int n, const vector<Edge> &edges, int s, int t,
22                             int k) {
23     vector<vector<int>> g(n);
24     for (int i = 0; i < ssize(edges); i += 1) { g[edges[i].u].push_back(i); }
25     vector<int> par(n, -1), p;
26     vector<i64> d(n, -1);
27     minimum_heap<pair<i64, int>> pq;
28     pq.push({d[s] = 0, s});
29     while (not pq.empty()) {
30         auto [du, u] = pq.top();
31         pq.pop();
32         if (du > d[u]) { continue; }
33         p.push_back(u);

```

```

34     for (int i : g[u]) {
35         auto [_, v, w] = edges[i];
36         if (d[v] == -1 or d[v] > d[u] + w) {
37             par[v] = i;
38             pq.push({d[v] = d[u] + w, v});
39         }
40     }
41 }
42 if (d[t] == -1) { return vector<i64>(k, -1); }
43 vector<Node *> heap(n);
44 for (int i = 0; i < ssize(edges); i += 1) {
45     auto [u, v, w] = edges[i];
46     if (~d[u] and ~d[v] and par[v] != i) {
47         heap[v] = merge(heap[v], new Node(u, d[u] + w - d[v]));
48     }
49 }
50 for (int u : p) {
51     if (u != s) { heap[u] = merge(heap[u], heap[edges[par[u]].u]); }
52 }
53 minimum_heap<pair<i64, Node *>> q;
54 if (heap[t]) { q.push({d[t] + heap[t]->w, heap[t]}); }
55 vector<i64> res = {d[t]};
56 for (int i = 1; i < k and not q.empty(); i += 1) {
57     auto [w, p] = q.top();
58     q.pop();
59     res.push_back(w);
60     if (heap[p->v]) { q.push({w + heap[p->v]->w, heap[p->v]}); }
61     for (auto c : {p->l, p->r}) {
62         if (c) { q.push({w + c->w - p->w, c}); }
63     }
64 }
65 res.resize(k, -1);
66 return res;
67 }

```

## 2.6 Global Minimum Cut

```

1 i64 global_minimum_cut(vector<vector<i64>> &w) {
2     int n = w.size();
3     if (n == 2) { return w[0][1]; }
4     vector<bool> in(n);
5     vector<int> add;
6     vector<i64> s(n);
7     i64 st = 0;
8     for (int i = 0; i < n; i += 1) {
9         int k = -1;
10        for (int j = 0; j < n; j += 1) {
11            if (not in[j]) {
12                if (k == -1 or s[j] > s[k]) { k = j; }
13            }
14        }
15        add.push_back(k);

```

```

16     st = s[k];
17     in[k] = true;
18     for (int j = 0; j < n; j += 1) { s[j] += w[j][k]; }
19 }
20 for (int i = 0; i < n; i += 1) {}
21 int x = add.rbegin()[1], y = add.back();
22 if (x == n - 1) { swap(x, y); }
23 for (int i = 0; i < n; i += 1) {
24     swap(w[y][i], w[n - 1][i]);
25     swap(w[i][y], w[i][n - 1]);
26 }
27 for (int i = 0; i + 1 < n; i += 1) {
28     w[i][x] += w[i][n - 1];
29     w[x][i] += w[n - 1][i];
30 }
31 w.pop_back();
32 return min(st, stoer_wagner(w));
33 }

```

## 2.7 Minimum Perfect Matching on Bipartite Graph

```

1 minimum_perfect_matching_on_bipartite_graph(const vector<vector<i64>>& w) {
2     i64 n = w.size();
3     vector<int> rm(n, -1), cm(n, -1);
4     vector<i64> pi(n);
5     auto resid = [&](int r, int c) { return w[r][c] - pi[c]; };
6     for (int c = 0; c < n; c += 1) {
7         int r =
8             ranges::min(views::iota(0, n), {}, [&](int r) { return w[r][c]; });
9         pi[c] = w[r][c];
10        if (rm[r] == -1) {
11            rm[r] = c;
12            cm[c] = r;
13        }
14    }
15    vector<int> cols(n);
16    iota(cols.begin(), cols.end(), 0);
17    for (int r = 0; r < n; r += 1) {
18        if (rm[r] != -1) { continue; }
19        vector<i64> d(n);
20        for (int c = 0; c < n; c += 1) { d[c] = resid(r, c); }
21        vector<int> pre(n, r);
22        int scan = 0, label = 0, last = 0, col = -1;
23        [&]() {
24            while (true) {
25                if (scan == label) {
26                    last = scan;
27                    i64 min = d[cols[scan]];
28                    for (int j = scan; j < n; j += 1) {
29                        int c = cols[j];
30                        if (d[c] <= min) {
31                            if (d[c] < min) {

```

```

32                        min = d[c];
33                        label = scan;
34                    }
35                    swap(cols[j], cols[label++]);
36                }
37            }
38            for (int j = scan; j < label; j += 1) {
39                if (int c = cols[j]; cm[c] == -1) {
40                    col = c;
41                    return;
42                }
43            }
44        }
45        int c1 = cols[scan++], r1 = cm[c1];
46        for (int j = label; j < n; j += 1) {
47            int c2 = cols[j];
48            i64 len = resid(r1, c2) - resid(r1, c1);
49            if (d[c2] > d[c1] + len) {
50                d[c2] = d[c1] + len;
51                pre[c2] = r1;
52                if (len == 0) {
53                    if (cm[c2] == -1) {
54                        col = c2;
55                        return;
56                    }
57                    swap(cols[j], cols[label++]);
58                }
59            }
60        }
61    }
62 }();
63 for (int i = 0; i < last; i += 1) {
64     int c = cols[i];
65     pi[c] += d[c] - d[col];
66 }
67 for (int t = col; t != -1;) {
68     col = t;
69     int r = pre[col];
70     cm[col] = r;
71     swap(rm[r], t);
72 }
73 }
74 i64 res = 0;
75 for (int i = 0; i < n; i += 1) { res += w[i][rm[i]]; }
76 return {res, rm};
77 }

```

## 2.8 Matching on General Graph

```

1 vector<int> matching(const vector<vector<int>> &g) {
2     int n = g.size();
3     int mark = 0;

```



```

4  vector<int> matched(n, -1), par(n, -1), book(n);
5  auto match = [&](int s) {
6      vector<int> c(n), type(n, -1);
7      iota(c.begin(), c.end(), 0);
8      queue<int> q;
9      q.push(s);
10     type[s] = 0;
11     while (not q.empty()) {
12         int u = q.front();
13         q.pop();
14         for (int v : g[u])
15             if (type[v] == -1) {
16                 par[v] = u;
17                 type[v] = 1;
18                 int w = matched[v];
19                 if (w == -1) {
20                     [&](int u) {
21                         while (u != -1) {
22                             int v = matched[par[u]];
23                             matched[matched[u] = par[u]] = u;
24                             u = v;
25                         }
26                     }(v);
27                     return;
28                 }
29                 q.push(w);
30                 type[w] = 0;
31             } else if (not type[v] and c[u] != c[v]) {
32                 int w = [&](int u, int v) {
33                     mark += 1;
34                     while (true) {
35                         if (u != -1) {
36                             if (book[u] == mark) { return u; }
37                             book[u] = mark;
38                             u = c[par[matched[u]]];
39                         }
40                         swap(u, v);
41                     }
42                 }(u, v);
43                 auto up = [&](int u, int v, int w) {
44                     while (c[u] != w) {
45                         par[u] = v;
46                         v = matched[u];
47                         if (type[v] == 1) {
48                             q.push(v);
49                             type[v] == 0;
50                         }
51                         if (c[u] == u) { c[u] = w; }
52                         if (c[v] == v) { c[v] = w; }
53                         u = par[v];
54                     }
55                 };
56                 up(u, v, w);

```

```

57         up(v, u, w);
58         for (int i = 0; i < n; i += 1) { c[i] = c[c[i]]; }
59     }
60 }
61 };
62 for (int i = 0; i < n; i += 1) {
63     if (matched[i] == -1) { match(i); }
64 }
65 return matched;
66 }

```

## 2.9 Maximum Flow

```

1  struct HighestLabelPreflowPush {
2      int n;
3      vector<vector<int>> g;
4      vector<Edge> edges;
5      HighestLabelPreflowPush(int n) : n(n), g(n) {}
6      int add(int u, int v, i64 f) {
7          if (u == v) { return -1; }
8          int i = ssize(edges);
9          edges.push_back({u, v, f});
10         g[u].push_back(i);
11         edges.push_back({v, u, 0});
12         g[v].push_back(i + 1);
13         return i;
14     }
15     i64 max_flow(int s, int t) {
16         vector<i64> p(n);
17         vector<int> h(n), cur(n), count(n * 2);
18         vector<vector<int>> pq(n * 2);
19         auto push = [&](int i, i64 f) {
20             auto [u, v, _] = edges[i];
21             if (not p[v] and f) { pq[h[v]].push_back(v); }
22             edges[i].f -= f;
23             edges[i ^ 1].f += f;
24             p[u] -= f;
25             p[v] += f;
26         };
27         h[s] = n;
28         count[0] = n - 1;
29         p[t] = 1;
30         for (int i : g[s]) { push(i, edges[i].f); }
31         for (int hi = 0;;) {
32             while (pq[hi].empty()) {
33                 if (not hi--) { return -p[s]; }
34             }
35             int u = pq[hi].back();
36             pq[hi].pop_back();
37             while (p[u] > 0) {
38                 if (cur[u] == ssize(g[u])) {
39                     h[u] = n * 2 + 1;

```

```

40     for (int i = 0; i < ssize(g[u]); i += 1) {
41         auto [_ , v, f] = edges[g[u][i]];
42         if (f and h[u] > h[v] + 1) {
43             h[u] = h[v] + 1;
44             cur[u] = i;
45         }
46     }
47     count[h[u]] += 1;
48     if (not(count[hi] -= 1) and hi < n) {
49         for (int i = 0; i < n; i += 1) {
50             if (h[i] > hi and h[i] < n) {
51                 count[h[i]] -= 1;
52                 h[i] = n + 1;
53             }
54         }
55     }
56     hi = h[u];
57 } else {
58     int i = g[u][cur[u]];
59     auto [_ , v, f] = edges[i];
60     if (f and h[u] == h[v] + 1) {
61         push(i, min(p[u], f));
62     } else {
63         cur[u] += 1;
64     }
65 }
66 }
67 }
68 return i64(0);
69 }
70 };
71
72 struct Dinic {
73     int n;
74     vector<vector<int>>> g;
75     vector<Edge> edges;
76     vector<int> level;
77     Dinic(int n) : n(n), g(n) {}
78     int add(int u, int v, i64 f) {
79         if (u == v) { return -1; }
80         int i = ssize(edges);
81         edges.push_back({u, v, f});
82         g[u].push_back(i);
83         edges.push_back({v, u, 0});
84         g[v].push_back(i + 1);
85         return i;
86     }
87     i64 max_flow(int s, int t) {
88         i64 flow = 0;
89         queue<int> q;
90         vector<int> cur;
91         auto bfs = [&]() {
92             level.assign(n, -1);

```

```

93         level[s] = 0;
94         q.push(s);
95         while (not q.empty()) {
96             int u = q.front();
97             q.pop();
98             for (int i : g[u]) {
99                 auto [_ , v, c] = edges[i];
100                 if (c and level[v] == -1) {
101                     level[v] = level[u] + 1;
102                     q.push(v);
103                 }
104             }
105         }
106         return ~level[t];
107     };
108     auto dfs = [&](auto& dfs, int u, i64 limit) -> i64 {
109         if (u == t) { return limit; }
110         i64 res = 0;
111         for (int& i = cur[u]; i < ssize(g[u]) and limit; i += 1) {
112             int j = g[u][i];
113             auto [_ , v, f] = edges[j];
114             if (level[v] == level[u] + 1 and f) {
115                 if (i64 d = dfs(dfs, v, min(f, limit)); d) {
116                     limit -= d;
117                     res += d;
118                     edges[j].f -= d;
119                     edges[j ^ 1].f += d;
120                 }
121             }
122         }
123         return res;
124     };
125     while (bfs()) {
126         cur.assign(n, 0);
127         while (i64 f = dfs(dfs, s, numeric_limits<i64>::max())) { flow += f; }
128     }
129     return flow;
130 }
131 };

```

## 2.10 Minimum Cost Maximum Flow

Constraints: there is no edge with negative cost.

```

1 struct MinimumCostMaximumFlow {
2     template <typename T>
3     using minimum_heap = priority_queue<T, vector<T>, greater<T>>>;
4     int n;
5     vector<Edge> edges;
6     vector<vector<int>>> g;
7     MinimumCostMaximumFlow(int n) : n(n), g(n) {}
8     int add_edge(int u, int v, i64 f, i64 c) {
9         int i = edges.size();

```

```

10 edges.push_back({u, v, f, c});
11 edges.push_back({v, u, 0, -c});
12 g[u].push_back(i);
13 g[v].push_back(i + 1);
14 return i;
15 }
16 pair<i64, i64> flow(int s, int t) {
17     constexpr i64 inf = numeric_limits<i64>::max();
18     vector<i64> d, h(n);
19     vector<int> p;
20     auto dijkstra = [&]() {
21         d.assign(n, inf);
22         p.assign(n, -1);
23         minimum_heap<pair<i64, int>> q;
24         q.emplace(d[s] = 0, s);
25         while (not q.empty()) {
26             auto [du, u] = q.top();
27             q.pop();
28             if (du > d[u]) { continue; }
29             for (int i : g[u]) {
30                 auto [_, v, f, c] = edges[i];
31                 if (f and d[v] > d[u] + h[u] - h[v] + c) {
32                     p[v] = i;
33                     q.emplace(d[v] = d[u] + h[u] - h[v] + c, v);
34                 }
35             }
36         }
37         return ~p[t];
38     };
39     i64 f = 0, c = 0;
40     while (dijkstra()) {
41         for (int i = 0; i < n; i += 1) { h[i] += d[i]; }
42         vector<int> path;
43         for (int u = t; u != s; u = edges[p[u]].u) { path.push_back(p[u]); }
44         i64 mf =
45             edges[ranges::min(path, {}, [&](int i) { return edges[i].f; } )].f;
46         f += mf;
47         c += mf * h[t];
48         for (int i : path) {
49             edges[i].f -= mf;
50             edges[i ^ 1].f += mf;
51         }
52     }
53     return {f, c};
54 }
55 };

```

## 3 Data Structure

### 3.1 Disjoint Set Union

```

1 struct DisjointSetUnion {
2     vector<int> dsu;
3     DisjointSetUnion(int n) : dsu(n, -1) {}
4     int find(int u) { return dsu[u] < 0 ? u : dsu[u] = find(dsu[u]); }
5     void merge(int u, int v) {
6         u = find(u);
7         v = find(v);
8         if (u != v) {
9             if (dsu[u] > dsu[v]) { swap(u, v); }
10            dsu[u] += dsu[v];
11            dsu[v] = u;
12        }
13    }
14 };
15 struct RollbackDisjointSetUnion {
16     vector<pair<int, int>> stack;
17     vector<int> dsu;
18     RollbackDisjointSetUnion(int n) : dsu(n, -1) {}
19     int find(int u) { return dsu[u] < 0 ? u : find(dsu[u]); }
20     int time() { return ssize(stack); }
21     bool merge(int u, int v) {
22         if ((u = find(u)) == (v = find(v))) { return false; }
23         if (dsu[u] < dsu[v]) { swap(u, v); }
24         stack.emplace_back(u, dsu[u]);
25         dsu[v] += dsu[u];
26         dsu[u] = v;
27         return true;
28     }
29     void rollback(int t) {
30         while (ssize(stack) > t) {
31             auto [u, dsu_u] = stack.back();
32             stack.pop_back();
33             dsu[dsu[u]] -= dsu_u;
34             dsu[u] = dsu_u;
35         }
36     }
37 };

```

### 3.2 Sparse Table

```

1 struct SparseTable {
2     vector<vector<int>> table;
3     SparseTable() {}
4     SparseTable(const vector<int> &a) {
5         int n = a.size(), h = bit_width(a.size());
6         table.resize(h);
7         table[0] = a;
8         for (int i = 1; i < h; i += 1) {
9             table[i].resize(n - (1 << i) + 1);
10            for (int j = 0; j + (1 << i) <= n; j += 1) {
11                table[i][j] = min(table[i - 1][j], table[i - 1][j + (1 << (i - 1))]);

```

```

12     }
13 }
14 }
15 int query(int l, int r) {
16     int h = bit_width(unsigned(r - 1)) - 1;
17     return min(table[h][l], table[h][r - (1 << h)]);
18 }
19 };
20 struct DisjointSparseTable {
21     vector<vector<int>> table;
22     DisjointSparseTable(const vector<int> &a) {
23         int h = bit_width(a.size() - 1), n = a.size();
24         table.resize(h, a);
25         for (int i = 0; i < h; i += 1) {
26             for (int j = 0; j + (1 << i) < n; j += (2 << i)) {
27                 for (int k = j + (1 << i) - 2; k >= j; k -= 1) {
28                     table[i][k] = min(table[i][k], table[i][k + 1]);
29                 }
30                 for (int k = j + (1 << i) + 1; k < j + (2 << i) and k < n; k += 1) {
31                     table[i][k] = min(table[i][k], table[i][k - 1]);
32                 }
33             }
34         }
35     }
36     int query(int l, int r) {
37         if (l + 1 == r) { return table[0][l]; }
38         int i = bit_width(unsigned(l ^ (r - 1))) - 1;
39         return min(table[i][l], table[i][r - 1]);
40     }
41 };

```

### 3.3 Treap

```

1 struct Node {
2     static constexpr bool persistent = true;
3     static mt19937_64 mt;
4     Node *l, *r;
5     u64 priority;
6     int size, v;
7     i64 sum;
8     Node(const Node &other) { memcpy(this, &other, sizeof(Node)); }
9     Node(int v) : v(v), sum(v), priority(mt()), size(1) { l = r = nullptr; }
10    Node *update(Node *l, Node *r) {
11        Node *p = persistent ? new Node(*this) : this;
12        p->l = l;
13        p->r = r;
14        p->size = (l ? l->size : 0) + 1 + (r ? r->size : 0);
15        p->sum = (l ? l->sum : 0) + v + (r ? r->sum : 0);
16        return p;
17    }
18 };
19 mt19937_64 Node::mt;

```

```

20 pair<Node *, Node *> split_by_v(Node *p, int v) {
21     if (not p) { return {}; }
22     if (p->v < v) {
23         auto [l, r] = split_by_v(p->r, v);
24         return {p->update(p->l, l), r};
25     }
26     auto [l, r] = split_by_v(p->l, v);
27     return {l, p->update(r, p->r)};
28 }
29 pair<Node *, Node *> split_by_size(Node *p, int size) {
30     if (not p) { return {}; }
31     int l_size = p->l ? p->l->size : 0;
32     if (l_size < size) {
33         auto [l, r] = split_by_size(p->r, size - l_size - 1);
34         return {p->update(p->l, l), r};
35     }
36     auto [l, r] = split_by_size(p->l, size);
37     return {l, p->update(r, p->r)};
38 }
39 Node *merge(Node *l, Node *r) {
40     if (not l or not r) { return l ? r; }
41     if (l->priority < r->priority) { return r->update(merge(l, r->l), r->r); }
42     return l->update(l->l, merge(l->r, r));
43 }

```

### 3.4 Lines Maximum

```

1 struct Line {
2     mutable i64 k, b, p;
3     bool operator<(const Line& rhs) const { return k < rhs.k; }
4     bool operator<(const i64& x) const { return p < x; }
5 };
6 struct Lines : multiset<Line, less<>> {
7     static constexpr i64 inf = numeric_limits<i64>::max();
8     static i64 div(i64 a, i64 b) { return a / b - ((a ^ b) < 0 and a % b); }
9     bool isect(iterator x, iterator y) {
10         if (y == end()) { return x->p == inf, false; }
11         if (x->k == y->k) {
12             x->p = x->b > y->b ? inf : -inf;
13         } else {
14             x->p = div(y->b - x->b, x->k - y->k);
15         }
16         return x->p >= y->p;
17     }
18     void add(i64 k, i64 b) {
19         auto z = insert({k, b, 0}), y = z++, x = y;
20         while (isect(y, z)) { z = erase(z); }
21         if (x != begin() and isect(--x, y)) { isect(x, y = erase(y)); }
22         while ((y = x) != begin() and (--x)->p >= y->p) { isect(x, erase(y)); }
23     }
24     optional<i64> get(i64 x) {
25         if (empty()) { return {}; }

```

```

26     auto it = lower_bound(x);
27     return it->k * x + it->b;
28 }
29 };

```

### 3.5 Segments Maximum

```

1 struct Segment {
2     i64 k, b;
3     i64 get(i64 x) { return k * x + b; }
4 };
5 struct Segments {
6     struct Node {
7         optional<Segment> s;
8         Node *l, *r;
9     };
10    i64 tl, tr;
11    Node *root;
12    Segments(i64 tl, i64 tr) : tl(tl), tr(tr), root(nullptr) {}
13    void add(i64 l, i64 r, i64 k, i64 b) {
14        function<void(Node *&, i64, i64, Segment)> rec = [&](Node *&p, i64 tl,
15                                                    i64 tr, Segment s) {
16            if (p == nullptr) { p = new Node(); }
17            i64 tm = midpoint(tl, tr);
18            if (tl >= l and tr <= r) {
19                if (not p->s) {
20                    p->s = s;
21                    return;
22                }
23                auto t = p->s.value();
24                if (t.get(tl) >= s.get(tl)) {
25                    if (t.get(tr) >= s.get(tr)) { return; }
26                    if (t.get(tm) >= s.get(tm)) { return rec(p->r, tm + 1, tr, s); }
27                    p->s = s;
28                    return rec(p->l, tl, tm, t);
29                }
30                if (t.get(tr) <= s.get(tr)) {
31                    p->s = s;
32                    return;
33                }
34                if (t.get(tm) <= s.get(tm)) {
35                    p->s = s;
36                    return rec(p->r, tm + 1, tr, t);
37                }
38                return rec(p->l, tl, tm, s);
39            }
40            if (l <= tm) { rec(p->l, tl, tm, s); }
41            if (r > tm) { rec(p->r, tm + 1, tr, s); }
42        };
43        rec(root, tl, tr, {k, b});
44    }
45    optional<i64> get(i64 x) {

```

```

46        optional<i64> res = {};
47        function<void(Node *, i64, i64)> rec = [&](Node *p, i64 tl, i64 tr) {
48            if (p == nullptr) { return; }
49            i64 tm = midpoint(tl, tr);
50            if (p->s) {
51                i64 y = p->s.value().get(x);
52                if (not res or res.value() < y) { res = y; }
53            }
54            if (x <= tm) {
55                rec(p->l, tl, tm);
56            } else {
57                rec(p->r, tm + 1, tr);
58            }
59        };
60        rec(root, tl, tr);
61        return res;
62    }
63 };

```

### 3.6 Segment Beats

```

1 struct Mv {
2     static constexpr i64 inf = numeric_limits<i64>::max() / 2;
3     i64 mv, smv, cmv, tmv;
4     bool less;
5     i64 def() { return less ? inf : -inf; }
6     i64 mmv(i64 x, i64 y) { return less ? min(x, y) : max(x, y); }
7     Mv(i64 x, bool less) : less(less) {
8         mv = x;
9         smv = tmv = def();
10        cmv = 1;
11    }
12    void up(const Mv& ls, const Mv& rs) {
13        mv = mmv(ls.mv, rs.mv);
14        smv = mmv(ls.mv == mv ? ls.smv : ls.mv, rs.mv == mv ? rs.smv : rs.mv);
15        cmv = (ls.mv == mv ? ls.cmv : 0) + (rs.mv == mv ? rs.cmv : 0);
16    }
17    void add(i64 x) {
18        mv += x;
19        if (smv != def()) { smv += x; }
20        if (tmv != def()) { tmv += x; }
21    }
22 };
23 struct Node {
24     Mv mn, mx;
25     i64 sum, tsum;
26     Node *ls, *rs;
27     Node(i64 x = 0) : sum(x), tsum(0), mn(x, true), mx(x, false) {
28         ls = rs = nullptr;
29     }
30    void up() {
31        sum = ls->sum + rs->sum;

```

```

32     mx.up(ls->mx, rs->mx);
33     mn.up(ls->mn, rs->mn);
34 }
35 void down(int tl, int tr) {
36     if (tsum) {
37         int tm = midpoint(tl, tr);
38         ls->add(tl, tm, tsum);
39         rs->add(tm, tr, tsum);
40         tsum = 0;
41     }
42     if (mn.tmv != mn.def()) {
43         ls->ch(mn.tmv, true);
44         rs->ch(mn.tmv, true);
45         mn.tmv = mn.def();
46     }
47     if (mx.tmv != mx.def()) {
48         ls->ch(mx.tmv, false);
49         rs->ch(mx.tmv, false);
50         mx.tmv = mx.def();
51     }
52 }
53 bool cmp(i64 x, i64 y, bool less) { return less ? x < y : x > y; }
54 void add(int tl, int tr, i64 x) {
55     sum += (tr - tl) * x;
56     tsum += x;
57     mx.add(x);
58     mn.add(x);
59 }
60 void ch(i64 x, bool less) {
61     auto &lhs = less ? mn : mx, &rhs = less ? mx : mn;
62     if (not cmp(x, rhs.mv, less)) { return; }
63     sum += (x - rhs.mv) * rhs.cmv;
64     if (lhs.smv == rhs.mv) { lhs.smv = x; }
65     if (lhs.mv == rhs.mv) { lhs.mv = x; }
66     if (cmp(x, rhs.tmv, less)) { rhs.tmv = x; }
67     rhs.mv = lhs.tmv = x;
68 }
69 void add(int tl, int tr, int l, int r, i64 x) {
70     if (tl >= l and tr <= r) { return add(tl, tr, x); }
71     down(tl, tr);
72     int tm = midpoint(tl, tr);
73     if (l < tm) { ls->add(tl, tm, l, r, x); }
74     if (r > tm) { rs->add(tm, tr, l, r, x); }
75     up();
76 }
77 void ch(int tl, int tr, int l, int r, i64 x, bool less) {
78     auto &lhs = less ? mn : mx, &rhs = less ? mx : mn;
79     if (not cmp(x, rhs.mv, less)) { return; }
80     if (tl >= l and tr <= r and cmp(rhs.smv, x, less)) {
81         return ch(x, less);
82     }
83     down(tl, tr);
84     int tm = midpoint(tl, tr);

```

```

85     if (l < tm) { ls->ch(tl, tm, l, r, x, less); }
86     if (r > tm) { rs->ch(tm, tr, l, r, x, less); }
87     up();
88 }
89 i64 get(int tl, int tr, int l, int r) {
90     if (tl >= l and tr <= r) { return sum; }
91     down(tl, tr);
92     i64 res = 0;
93     int tm = midpoint(tl, tr);
94     if (l < tm) { res += ls->get(tl, tm, l, r); }
95     if (r > tm) { res += rs->get(tm, tr, l, r); }
96     return res;
97 }
98 };

```

## 3.7 Tree

### 3.7.1 Least Common Ancestor

```

1 struct LeastCommonAncestor {
2     SparseTable st;
3     vector<int> p, time, a, par;
4     LeastCommonAncestor(int root, const vector<vector<int>> &g) {
5         int n = g.size();
6         time.resize(n, -1);
7         par.resize(n, -1);
8         function<void(int)> dfs = [&](int u) {
9             time[u] = p.size();
10            p.push_back(u);
11            for (int v : g[u]) {
12                if (time[v] == -1) {
13                    par[v] = u;
14                    dfs(v);
15                }
16            }
17        };
18        dfs(root);
19        a.resize(n);
20        for (int i = 1; i < n; i += 1) { a[i] = time[par[p[i]]]; }
21        st = SparseTable(a);
22    }
23    int query(int u, int v) {
24        if (u == v) { return u; }
25        if (time[u] > time[v]) { swap(u, v); }
26        return p[st.query(time[u] + 1, time[v] + 1)];
27    }
28 };

```

### 3.7.2 Link Cut Tree

```

1 template <class T, class E, class REV, class OP> struct Node {
2     T t, st;
3     bool reversed;
4     Node* par;
5     array<Node*, 2> ch;
6     Node(T t = E()) : t(t), st(t), reversed(false), par(nullptr) {
7         ch.fill(nullptr);
8     }
9     int get_s() {
10         if (par == nullptr) { return -1; }
11         if (par->ch[0] == this) { return 0; }
12         if (par->ch[1] == this) { return 1; }
13         return -1;
14     }
15     void push_up() {
16         st = OP()(ch[0] ? ch[0]->st : E(), OP()(t, ch[1] ? ch[1]->st : E()));
17     }
18     void reverse() {
19         reversed ^= 1;
20         st = REV()(st);
21     }
22     void push_down() {
23         if (reversed) {
24             swap(ch[0], ch[1]);
25             if (ch[0]) { ch[0]->reverse(); }
26             if (ch[1]) { ch[1]->reverse(); }
27             reversed = false;
28         }
29     }
30     void attach(int s, Node* u) {
31         if ((ch[s] = u)) { u->par = this; }
32         push_up();
33     }
34     void rotate() {
35         auto p = par;
36         auto pp = p->par;
37         int s = get_s();
38         int ps = p->get_s();
39         p->attach(s, ch[s ^ 1]);
40         attach(s ^ 1, p);
41         if (~ps) { pp->attach(ps, this); }
42         par = pp;
43     }
44     void splay() {
45         push_down();
46         while (~get_s() and ~par->get_s()) {
47             par->par->push_down();
48             par->push_down();
49             push_down();
50             (get_s() == par->get_s() ? par : this)->rotate();
51             rotate();
52         }
53         if (~get_s()) {

```

```

54         par->push_down();
55         push_down();
56         rotate();
57     }
58 }
59 void access() {
60     splay();
61     attach(1, nullptr);
62     while (par != nullptr) {
63         auto p = par;
64         p->splay();
65         p->attach(1, this);
66         rotate();
67     }
68 }
69 void make_root() {
70     access();
71     reverse();
72     push_down();
73 }
74 void link(Node* u) {
75     u->make_root();
76     access();
77     attach(1, u);
78 }
79 void cut(Node* u) {
80     u->make_root();
81     access();
82     if (ch[0] == u) {
83         ch[0] = u->par = nullptr;
84         push_up();
85     }
86 }
87 void set(T t) {
88     access();
89     this->t = t;
90     push_up();
91 }
92 T query(Node* u) {
93     u->make_root();
94     access();
95     return st;
96 }
97 };

```

## 4 String

### 4.1 Z

```

1 vector<int> fz(const string &s) {
2     int n = s.size();

```

```

3   vector<int> z(n);
4   for (int i = 1, j = 0; i < n; i += 1) {
5       z[i] = max(min(z[i - j], j + z[j] - i), 0);
6       while (i + z[i] < n and s[i + z[i]] == s[z[i]]) { z[i] += 1; }
7       if (i + z[i] > j + z[j]) { j = i; }
8   }
9   return z;
10 }

```

## 4.2 Lyndon Factorization

```

1 vector<int> lyndon_factorization(string const &s) {
2     vector<int> res = {0};
3     for (int i = 0, n = s.size(); i < n;) {
4         int j = i + 1, k = i;
5         for (; j < n and s[k] <= s[j]; j += 1) { k = s[k] < s[j] ? i : k + 1; }
6         while (i <= k) { res.push_back(i += j - k); }
7     }
8     return res;
9 }

```

## 4.3 Border

```

1 vector<int> fborder(const string &s) {
2     int n = s.size();
3     vector<int> res(n);
4     for (int i = 1; i < n; i += 1) {
5         int &j = res[i] = res[i - 1];
6         while (j and s[i] != s[j]) { j = res[j - 1]; }
7         j += s[i] == s[j];
8     }
9     return res;
10 }

```

## 4.4 Manacher

```

1 vector<int> manacher(const string &s) {
2     int n = s.size();
3     vector<int> p(n);
4     for (int i = 0, j = 0; i < n; i += 1) {
5         if (j + p[j] > i) { p[i] = min(p[j * 2 - i], j + p[j] - i); }
6         while (i >= p[i] and i + p[i] < n and s[i - p[i]] == s[i + p[i]]) {
7             p[i] += 1;
8         }
9         if (i + p[i] > j + p[j]) { j = i; }
10    }
11    return p;
12 }

```

## 4.5 Suffix Array

```

1 pair<vector<int>, vector<int>> binary_lifting(const string &s) {
2     int n = s.size(), k = 0;
3     vector<int> p(n), rank(n), q, count;
4     iota(p.begin(), p.end(), 0);
5     ranges::sort(p, {}, [&](int i) { return s[i]; });
6     for (int i = 0; i < n; i += 1) {
7         rank[p[i]] = i and s[p[i]] == s[p[i - 1]] ? rank[p[i - 1]] : k++;
8     }
9     for (int m = 1; m < n; m *= 2) {
10        q.resize(m);
11        iota(q.begin(), q.end(), n - m);
12        for (int i : p) {
13            if (i >= m) { q.push_back(i - m); }
14        }
15        count.assign(k, 0);
16        for (int i : rank) { count[i] += 1; }
17        partial_sum(count.begin(), count.end(), count.begin());
18        for (int i = n - 1; i >= 0; i -= 1) { p[count[rank[q[i]]] -= 1] = q[i]; }
19        auto previous = rank;
20        previous.resize(2 * n, -1);
21        k = 0;
22        for (int i = 0; i < n; i += 1) {
23            rank[p[i]] = i and previous[p[i]] == previous[p[i - 1]] and
24                           previous[p[i] + m] == previous[p[i - 1] + m]
25                           ? rank[p[i - 1]]
26                           : k++;
27        }
28    }
29    vector<int> lcp(n);
30    k = 0;
31    for (int i = 0; i < n; i += 1) {
32        if (rank[i]) {
33            k = max(k - 1, 0);
34            int j = p[rank[i] - 1];
35            while (i + k < n and j + k < n and s[i + k] == s[j + k]) { k += 1; }
36            lcp[rank[i]] = k;
37        }
38    }
39    return {p, lcp};
40 }

```

## 4.6 Aho-Corasick Automaton

```

1 constexpr int sigma = 26;
2 struct Node {
3     int link;
4     array<int, sigma> next;
5     Node() : link(0) { next.fill(0); }
6 };

```



```

7 struct AhoCorasick : vector<Node> {
8     AhoCorasick() : vector<Node>(1) {}
9     int add(const string &s, char first = 'a') {
10         int p = 0;
11         for (char si : s) {
12             int c = si - first;
13             if (not at(p).next[c]) {
14                 at(p).next[c] = size();
15                 emplace_back();
16             }
17             p = at(p).next[c];
18         }
19         return p;
20     }
21     void init() {
22         queue<int> q;
23         for (int i = 0; i < sigma; i += 1) {
24             if (at(0).next[i]) { q.push(at(0).next[i]); }
25         }
26         while (not q.empty()) {
27             int u = q.front();
28             q.pop();
29             for (int i = 0; i < sigma; i += 1) {
30                 if (at(u).next[i]) {
31                     at(at(u).next[i]).link = at(at(u).link).next[i];
32                     q.push(at(u).next[i]);
33                 } else {
34                     at(u).next[i] = at(at(u).link).next[i];
35                 }
36             }
37         }
38     }
39 };

```

```

17         at(p).next[c] = clone;
18         p = at(p).link;
19     }
20     at(q).link = clone;
21     return clone;
22 }
23 int cur = size();
24 emplace_back();
25 back().len = at(p).len + 1;
26 while (~p and at(p).next[c] == -1) {
27     at(p).next[c] = cur;
28     p = at(p).link;
29 }
30 if (~p) {
31     int q = at(p).next[c];
32     if (at(p).len + 1 == at(q).len) {
33         back().link = q;
34     } else {
35         int clone = size();
36         push_back(at(q));
37         back().len = at(p).len + 1;
38         while (~p and at(p).next[c] == q) {
39             at(p).next[c] = clone;
40             p = at(p).link;
41         }
42         at(q).link = at(cur).link = clone;
43     }
44 } else {
45     back().link = 0;
46 }
47 return cur;
48 }
49 };

```

## 4.7 Suffix Automaton

```

1 struct Node {
2     int link, len;
3     array<int, sigma> next;
4     Node() : link(-1), len(0) { next.fill(-1); }
5 };
6 struct SuffixAutomaton : vector<Node> {
7     SuffixAutomaton() : vector<Node>(1) {}
8     int extend(int p, int c) {
9         if (~at(p).next[c]) {
10             // For online multiple strings.
11             int q = at(p).next[c];
12             if (at(p).len + 1 == at(q).len) { return q; }
13             int clone = size();
14             push_back(at(q));
15             back().len = at(p).len + 1;
16             while (~p and at(p).next[c] == q) {

```

## 4.8 Palindromic Tree

```

1 struct Node {
2     int sum, len, link;
3     array<int, sigma> next;
4     Node(int len) : len(len) {
5         sum = link = 0;
6         next.fill(0);
7     }
8 };
9 struct PalindromicTree : vector<Node> {
10     int last;
11     vector<int> s;
12     PalindromicTree() : last(0) {
13         emplace_back(0);
14         emplace_back(-1);
15         at(0).link = 1;
16     }

```

```

17 int get_link(int u, int i) {
18     while (i < at(u).len + 1 or s[i - at(u).len - 1] != s[i]) u = at(u).link;
19     return u;
20 }
21 void extend(int i) {
22     int cur = get_link(last, i);
23     if (not at(cur).next[s[i]]) {
24         int now = size();
25         emplace_back(at(cur).len + 2);
26         back().link = at(get_link(at(cur).link, i)).next[s[i]];
27         back().sum = at(back().link).sum + 1;
28         at(cur).next[s[i]] = now;
29     }
30     last = at(cur).next[s[i]];
31 }
32 };

```

```

4 if (power(y, (p - 1) / 2, p) != 1) { return -1; }
5 uniform_int_distribution uid(i64(0), p - 1);
6 i64 x, w;
7 do {
8     x = uid(mt);
9     w = (x * x + p - y) % p;
10 } while (power(w, (p - 1) / 2, p) == 1);
11 auto mul = [&](pair<i64, i64> a, pair<i64, i64> b) {
12     return pair((a.first * b.first + a.second * b.second % p * w) % p,
13                 (a.first * b.second + a.second * b.first) % p);
14 };
15 pair<i64, i64> a = {x, 1}, res = {1, 0};
16 for (i64 r = (p + 1) >> 1; r; r >>= 1, a = mul(a, a)) {
17     if (r & 1) { res = mul(res, a); }
18 }
19 return res.first;
20 }

```

## 5 Number Theory

### 5.1 Gaussian Integer

```

1 i64 div_floor(i64 x, i64 y) { return x / y - (x % y < 0); }
2 i64 div_ceil(i64 x, i64 y) { return x / y + (x % y > 0); }
3 i64 div_round(i64 x, i64 y) { return div_floor(2 * x + y, 2 * y); }
4 struct Gauss {
5     i64 x, y;
6     i64 norm() { return x * x + y * y; }
7     bool operator!=(i64 r) { return y or x != r; }
8     Gauss operator~() { return {x, -y}; }
9     Gauss operator-(Gauss rhs) { return {x - rhs.x, y - rhs.y}; }
10    Gauss operator*(Gauss rhs) {
11        return {x * rhs.x - y * rhs.y, x * rhs.y + y * rhs.x};
12    }
13    Gauss operator/(Gauss rhs) {
14        auto [x, y] = operator*(~rhs);
15        return {div_round(x, rhs.norm()), div_round(y, rhs.norm())};
16    }
17    Gauss operator%(Gauss rhs) { return operator-(rhs*(operator/(rhs))); }
18 };

```

```

1 i64 log(i64 x, i64 y, i64 n) {
2     if (y == 1 or n == 1) { return 0; }
3     if (not x) { return y ? -1 : 1; }
4     i64 res = 0, k = 1 % n;
5     for (i64 d; k != y and (d = gcd(x, n)) != 1; res += 1) {
6         if (y % d) { return -1; }
7         n /= d;
8         y /= d;
9         k = k * (x / d) % n;
10    }
11    if (k == y) { return res; }
12    unordered_map<i64, i64> mp;
13    i64 px = 1, m = sqrt(n) + 1;
14    for (int i = 0; i < m; i += 1, px = px * x % n) { mp[y * px % n] = i; }
15    i64 ppx = k * px % n;
16    for (int i = 1; i <= m; i += 1, ppx = ppx * px % n) {
17        if (mp.count(ppx)) { return res + i * m - mp[ppx]; }
18    }
19    return -1;
20 }

```

### 5.2 Modular Arithmetic

#### 5.2.1 Sqrt

Find  $x$  such that  $x^2 \equiv y \pmod{p}$ .  
Constraints:  $p$  is prime and  $0 \leq y < p$ .

```

1 i64 sqrt(i64 y, i64 p) {
2     static mt19937_64 mt;
3     if (y <= 1) { return y; };

```

### 5.3 Chinese Remainder Theorem

```

1 tuple<i64, i64, i64> exgcd(i64 a, i64 b) {
2     i64 x = 1, y = 0, x1 = 0, y1 = 1;
3     while (b) {
4         i64 q = a / b;
5         tie(x, x1) = pair(x1, x - q * x1);

```

```

6     tie(y, y1) = pair(y1, y - q * y1);
7     tie(a, b) = pair(b, a - q * b);
8 }
9     return {a, x, y};
10 }
11 optional<pair<i64, i64>> linear_equations(i64 a0, i64 b0, i64 a1, i64 b1) {
12     auto [d, x, y] = exgcd(a0, a1);
13     if ((b1 - b0) % d) { return {}; }
14     i64 a = a0 / d * a1, b = (i128)(b1 - b0) / d * x % (a1 / d);
15     if (b < 0) { b += a1 / d; }
16     b = (i128)(a0 * b + b0) % a;
17     if (b < 0) { b += a; }
18     return {{a, b}};
19 }

```

## 5.4 Miller Rabin

```

1 bool miller_rabin(i64 n) {
2     static constexpr array<int, 9> p = {2, 3, 5, 7, 11, 13, 17, 19, 23};
3     if (n == 1) { return false; }
4     if (n == 2) { return true; }
5     if (not(n % 2)) { return false; }
6     int r = countr_zero(u64(n - 1));
7     i64 d = (n - 1) >> r;
8     for (int pi : p) {
9         if (pi >= n) { break; }
10        i64 x = power(pi, d, n);
11        if (x == 1 or x == n - 1) { continue; };
12        for (int j = 1; j < r; j += 1) {
13            x = (i128)x * x % n;
14            if (x == n - 1) { break; }
15        }
16        if (x != n - 1) { return false; }
17    }
18    return true;
19 };

```

## 5.5 Pollard Rho

```

1 vector<i64> pollard_rho(i64 n) {
2     static mt19937_64 mt;
3     uniform_int_distribution uid(i64(0), n);
4     if (n == 1) { return {}; }
5     vector<i64> res;
6     function<void(i64)> rho = [&](i64 n) {
7         if (miller_rabin(n)) { return res.push_back(n); }
8         i64 d = n;
9         while (d == n) {
10             d = 1;
11             for (i64 k = 1, y = 0, x = 0, s = 1, c = uid(mt); d == 1;

```

```

12         k <= 1, y = x, s = 1) {
13             for (int i = 1; i <= k; i += 1) {
14                 x = ((i128)x * x + c) % n;
15                 s = (i128)s * abs(x - y) % n;
16                 if (not(i % 127) or i == k) {
17                     d = gcd(s, n);
18                     if (d != 1) { break; }
19                 }
20             }
21         }
22     }
23     rho(d);
24     rho(n / d);
25 };
26 rho(n);
27 return res;
28 }

```

## 5.6 Primitive Root

Constraints:  $n = 2, 4, p^k, 2p^k$  where  $p$  is odd prime.

```

1 i64 phi(i64 n) {
2     auto pd = pollard_rho(n);
3     ranges::sort(pd);
4     pd.erase(ranges::unique(pd).begin(), pd.end());
5     for (i64 pi : pd) { n = n / pi * (pi - 1); }
6     return n;
7 }
8 i64 minimum_primitive_root(i64 n) {
9     i64 pn = phi(n);
10    auto pd = pollard_rho(pn);
11    ranges::sort(pd);
12    pd.erase(ranges::unique(pd).begin(), pd.end());
13    auto check = [&](i64 r) {
14        if (gcd(r, n) != 1) { return false; }
15        for (i64 pi : pd) {
16            if (power(r, pn / pi, n) == 1) { return false; }
17        }
18        return true;
19    };
20    i64 r = 1;
21    while (not check(r)) { r += 1; }
22    return r;
23 }

```

## 5.7 Sum of Floor

Returns  $\sum_{i=0}^{n-1} \lfloor \frac{ai+b}{m} \rfloor$ .

```

1 u64 sum_of_floor(u64 n, u64 m, u64 a, u64 b) {
2     u64 ans = 0;

```

```

3 while (n) {
4     ans += a / m * n * (n - 1) / 2;
5     a %= m;
6     ans += b / m * n;
7     b %= m;
8     u64 y = a * n + b;
9     if (y < m) { break; }
10    tie(n, m, a, b) = tuple(y / m, a, m, y % m);
11 }
12 return ans;
13 }

```

## 5.8 Minimum of Remainder

Returns  $\min\{(ai + b) \bmod m : 0 \leq i < n\}$ .

```

1 u64 min_of_mod(u64 n, u64 m, u64 a, u64 b, u64 c = 1, u64 p = 1, u64 q = 1) {
2     if (a == 0) { return b; }
3     if (c % 2) {
4         if (b >= a) {
5             u64 t = (m - b + a - 1) / a;
6             u64 d = (t - 1) * p + q;
7             if (n <= d) { return b; }
8             n -= d;
9             b += a * t - m;
10        }
11        b = a - 1 - b;
12    } else {
13        if (b < m - a) {
14            u64 t = (m - b - 1) / a;
15            u64 d = t * p;
16            if (n <= d) { return (n - 1) / p * a + b; }
17            n -= d;
18            b += a * t;
19        }
20        b = m - 1 - b;
21    }
22    u64 d = m / a;
23    u64 res = min_of_mod(n, a, m % a, b, c += 1, (d - 1) * p + q, d * p + q);
24    return c % 2 ? m - 1 - res : a - 1 - res;
25 }

```

## 5.9 Stern Brocot Tree

```

1 struct Node {
2     int a, b;
3     vector<pair<int, char>> p;
4     Node(int a, int b) : a(a), b(b) {
5         // gcd(a, b) == 1
6         while (a != 1 or b != 1) {
7             if (a > b) {

```

```

8         int k = (a - 1) / b;
9         p.emplace_back(k, 'R');
10        a -= k * b;
11    } else {
12        int k = (b - 1) / a;
13        p.emplace_back(k, 'L');
14        b -= k * a;
15    }
16 }
17 }
18 Node(vector<pair<int, char>> p, int _a = 1, int _b = 1)
19 : p(p), a(_a), b(_b) {
20     for (auto [c, d] : p | views::reverse) {
21         if (d == 'R') {
22             a += c * b;
23         } else {
24             b += c * a;
25         }
26     }
27 }
28 };

```

## 5.10 Nim Product

```

1 struct NimProduct {
2     array<array<u64, 64>, 64> mem;
3     NimProduct() {
4         for (int i = 0; i < 64; i += 1) {
5             for (int j = 0; j < 64; j += 1) {
6                 int k = i & j;
7                 if (k == 0) {
8                     mem[i][j] = u64(1) << (i | j);
9                 } else {
10                    int x = k & -k;
11                    mem[i][j] = mem[i ^ x][j] ^
12                               mem[(i ^ x) | (x - 1)][(j ^ x) | (i & (x - 1))];
13                }
14            }
15        }
16    }
17    u64 nim_product(u64 x, u64 y) {
18        u64 res = 0;
19        for (int i = 0; i < 64 and x >> i; i += 1) {
20            if ((x >> i) % 2) {
21                for (int j = 0; j < 64 and y >> j; j += 1) {
22                    if ((y >> j) % 2) { res ^= mem[i][j]; }
23                }
24            }
25        }
26        return res;
27    }
28 };

```

## 6 Numerical

### 6.1 Golden Search

```
1 template <int step> f64 golden_search(function<f64(f64)> f, f64 l, f64 r) {
2     f64 ml = (numbers::phi - 1) * l + (2 - numbers::phi) * r;
3     f64 mr = l + r - ml;
4     f64 fml = f(ml), fmr = f(mr);
5     for (int i = 0; i < step; i += 1)
6         if (fml > fmr) {
7             l = ml;
8             ml = mr;
9             fml = fmr;
10            fmr = f(mr = (numbers::phi - 1) * r + (2 - numbers::phi) * l);
11        } else {
12            r = mr;
13            mr = ml;
14            fmr = fml;
15            fml = f(ml = (numbers::phi - 1) * l + (2 - numbers::phi) * r);
16        }
17    return midpoint(l, r);
18 }
```

### 6.2 Adaptive Simpson

```
1 f64 simpson(function<f64(f64)> f, f64 l, f64 r) {
2     return (r - l) * (f(l) + f(r) + 4 * f(midpoint(l, r))) / 6;
3 }
4 f64 adaptive_simpson(const function<f64(f64)> &f, f64 l, f64 r, f64 eps) {
5     f64 m = midpoint(l, r);
6     f64 s = simpson(f, l, r);
7     f64 sl = simpson(f, l, m);
8     f64 sr = simpson(f, m, r);
9     f64 d = sl + sr - s;
10    if (abs(d) < 15 * eps) { return (sl + sr) + d / 15; }
11    return adaptive_simpson(f, l, m, eps / 2) +
12           adaptive_simpson(f, m, r, eps / 2);
13 }
```

### 6.3 Simplex

Returns maximum of  $cx$  s.t.  $ax \leq b$  and  $x \geq 0$ .

```
1 struct Simplex {
2     int n, m;
3     f64 z;
4     vector<vector<f64>> a;
5     vector<f64> b, c;
6     vector<int> base;
7     Simplex(int n, int m)
```

```
8     : n(n), m(m), a(m, vector<f64>(n)), b(m), c(n), base(n + m), z(0) {
9         iota(base.begin(), base.end(), 0);
10    }
11    void pivot(int out, int in) {
12        swap(base[out + n], base[in]);
13        f64 f = 1 / a[out][in];
14        for (f64 &aij : a[out]) { aij *= f; }
15        b[out] *= f;
16        a[out][in] = f;
17        for (int i = 0; i <= m; i += 1) {
18            if (i != out) {
19                auto &ai = i == m ? c : a[i];
20                f64 &bi = i == m ? z : b[i];
21                f64 f = -ai[in];
22                if (f < -eps or f > eps) {
23                    for (int j = 0; j < n; j += 1) { ai[j] += a[out][j] * f; }
24                    ai[in] = a[out][in] * f;
25                    bi += b[out] * f;
26                }
27            }
28        }
29    }
30    bool feasible() {
31        while (true) {
32            int i = ranges::min_element(b) - b.begin();
33            if (b[i] > -eps) { break; }
34            int k = -1;
35            for (int j = 0; j < n; j += 1) {
36                if (a[i][j] < -eps and (k == -1 or base[j] > base[k])) { k = j; }
37            }
38            if (k == -1) { return false; }
39            pivot(i, k);
40        }
41        return true;
42    }
43    bool bounded() {
44        while (true) {
45            int i = ranges::max_element(c) - c.begin();
46            if (c[i] < eps) { break; }
47            int k = -1;
48            for (int j = 0; j < m; j += 1) {
49                if (a[j][i] > eps) {
50                    if (k == -1) {
51                        k = j;
52                    } else {
53                        f64 d = b[j] * a[k][i] - b[k] * a[j][i];
54                        if (d < -eps or (d < eps and base[j] > base[k])) { k = j; }
55                    }
56                }
57            }
58            if (k == -1) { return false; }
59            pivot(k, i);
60        }
61    }
```

```

61     return true;
62 }
63 vector<f64> x() const {
64     vector<f64> res(n);
65     for (int i = n; i < n + m; i += 1) {
66         if (base[i] < n) { res[base[i]] = b[i - n]; }
67     }
68     return res;
69 }
70 };

```

## 6.4 Green's Theorem

$$\oint_C (Pdx + Qdy) = \iint_D \left( \frac{\partial Q}{\partial x} - \frac{\partial P}{\partial y} \right) dxdy.$$

## 6.5 Double Integral

$$\iint_D f(x, y) dxdy = \iint_D f(x(u, v), y(u, v)) \left| \begin{array}{cc} \frac{\partial x}{\partial u} & \frac{\partial x}{\partial v} \\ \frac{\partial y}{\partial u} & \frac{\partial y}{\partial v} \end{array} \right| dudv.$$

# 7 Convolution

## 7.1 Fast Fourier Transform on $\mathbb{C}$

```

1 void fft(vector<complex<f64>>& a, bool inverse) {
2     int n = a.size();
3     vector<int> r(n);
4     for (int i = 0; i < n; i += 1) {
5         r[i] = r[i / 2] / 2 | (i % 2 ? n / 2 : 0);
6     }
7     for (int i = 0; i < n; i += 1) {
8         if (i < r[i]) { swap(a[i], a[r[i]]); }
9     }
10    for (int m = 1; m < n; m *= 2) {
11        complex<f64> wn(exp((inverse ? 1.i : -1.i) * numbers::pi / (f64)m));
12        for (int i = 0; i < n; i += m * 2) {
13            complex<f64> w = 1;
14            for (int j = 0; j < m; j += 1, w = w * wn) {
15                auto &x = a[i + j + m], &y = a[i + j], t = w * x;
16                tie(x, y) = pair(y - t, y + t);
17            }
18        }
19    }
20    if (inverse) {
21        for (auto& ai : a) { ai /= n; }
22    }
23 }

```

## 7.2 Formal Power Series on $\mathbb{F}_p$

```

1 void fft(vector<i64>& a, bool inverse) {
2     int n = a.size();
3     vector<int> r(n);
4     for (int i = 0; i < n; i += 1) {
5         r[i] = r[i / 2] / 2 | (i % 2 ? n / 2 : 0);
6     }
7     for (int i = 0; i < n; i += 1) {
8         if (i < r[i]) { swap(a[i], a[r[i]]); }
9     }
10    for (int m = 1; m < n; m *= 2) {
11        i64 wn = power(inverse ? power(g, mod - 2) : g, (mod - 1) / m / 2);
12        for (int i = 0; i < n; i += m * 2) {
13            i64 w = 1;
14            for (int j = 0; j < m; j += 1, w = w * wn % mod) {
15                auto &x = a[i + j + m], &y = a[i + j], t = w * x % mod;
16                tie(x, y) = pair((y + mod - t) % mod, (y + t) % mod);
17            }
18        }
19    }
20    if (inverse) {
21        i64 inv = power(n, mod - 2);
22        for (auto& ai : a) { ai = ai * inv % mod; }
23    }
24 }

```

### 7.2.1 Newton's Method

$$h = g(f) \Leftrightarrow G(h) = f - g^{-1}(h) \equiv 0.$$

$$h = h_0 - \frac{G(h_0)}{G'(h_0)}.$$

### 7.2.2 Arithmetic

- For  $f = pg + q$ ,  $p^T = f^T g^T - 1$ .
- For  $h = \frac{1}{f}$ ,  $h = h_0(2 - h_0 f)$ .
- For  $h = \sqrt{f}$ ,  $h = \frac{1}{2}(h_0 + \frac{f}{h_0})$ .
- For  $h = \log f$ ,  $h = \int \frac{df}{f}$ .
- For  $h = \exp f$ ,  $h = h_0(1 + f - \log h_0)$ .

### 7.2.3 Interpolation

$$g(x) = \prod_i (x - x_i)$$

$$f(x) = \sum_{i=0}^{n-1} y_i \left( \prod_{j \neq i} \frac{x - x_j}{x_i - x_j} \right).$$

$$f(x) = \sum_{i=0}^{n-1} \frac{y_i}{g'(x_i)} \prod_{j \neq i} (x - x_j).$$

## 7.2.4 Primes with root 3

$$469762049 = 7 \times 2^{26} + 1.$$

$$4179340454199820289 = 29 \times 2^{57} + 1.$$

## 7.3 Circular Transform

$$A_{ij} = w_k^{ij}, A_{ij}^{-1} = \frac{1}{w_k} w_k^{-ij}.$$

## 7.4 Truncated Transform

$$\sum_{j=0}^{n-1} \frac{i}{\prod_{k=0}^j m_k} \bmod n \quad \text{for } 0 \leq i < \prod_{j=0}^{n-1} m_k.$$

# 8 Geometry

## 8.1 Pick's Theorem

$$\text{Area} = \#\{\text{points inside}\} + \frac{1}{2}\#\{\text{points on the border}\} - 1.$$

## 8.2 2D Geometry

P: point, L: line, G: convex hull or polygon, C: Circle.

```
1 template <typename T> T eps = 0;
2 template <> f64 eps<f64> = 1e-9;
3 template <typename T> int sign(T x) { return x < -eps<T> ? -1 : x > eps<T>; }
4 template <typename T> struct P {
5     T x, y;
6     explicit P(T x = 0, T y = 0) : x(x), y(y) {}
7     P operator*(T k) { return P(x * k, y * k); }
8     P operator+(P p) { return P(x + p.x, y + p.y); }
9     P operator-(P p) { return P(x - p.x, y - p.y); }
10    P operator-() { return P(-x, -y); }
11    T len2() { return x * x + y * y; }
12    T cross(P p) { return x * p.y - y * p.x; }
13    T dot(P p) { return x * p.x + y * p.y; }
14    bool operator==(P p) { return sign(x - p.x) == 0 and sign(y - p.y) == 0; }
15    int arg() { return y < 0 or (y == 0 and x > 0) ? -1 : x or y; }
16    P rotate90() { return P(-y, x); }
17 };
18 template <typename T> bool argument(P<T> lhs, P<T> rhs) {
19     if (lhs.arg() != rhs.arg()) { return lhs.arg() < rhs.arg(); }
20     return lhs.cross(rhs) > 0;
21 }
22 template <typename T> struct L {
23     P<T> a, b;
24     explicit L(P<T> a = {}, P<T> b = {}) : a(a), b(b) {}
25     P<T> v() { return b - a; }
26     bool contains(P<T> p) {
27         return sign((p - a).cross(p - b)) == 0 and sign((p - a).dot(p - b)) <= 0;
28     }
29 }
```

```
29 int left(P<T> p) { return sign(v().cross(p - a)); }
30 optional<pair<T, T>> intersection(L l) {
31     auto y = v().cross(l.v());
32     if (sign(y) == 0) { return {}; }
33     auto x = (l.a - a).cross(l.v());
34     return y < 0 ? pair(-x, -y) : pair(x, y);
35 }
36 };
37 template <typename T> struct G {
38     vector<P<T>> g;
39     explicit G(int n) : g(n) {}
40     explicit G(const vector<P<T>>& g) : g(g) {}
41     optional<int> winding(P<T> p) {
42         int n = g.size(), res = 0;
43         for (int i = 0; i < n; i += 1) {
44             auto a = g[i], b = g[(i + 1) % n];
45             L l(a, b);
46             if (l.contains(p)) { return {}; }
47             if (sign(l.v().y) < 0 and l.left(p) >= 0) { continue; }
48             if (sign(l.v().y) == 0) { continue; }
49             if (sign(l.v().y) > 0 and l.left(p) <= 0) { continue; }
50             if (sign(a.y - p.y) < 0 and sign(b.y - p.y) >= 0) { res += 1; }
51             if (sign(a.y - p.y) >= 0 and sign(b.y - p.y) < 0) { res -= 1; }
52         }
53         return res;
54     }
55     G convex() {
56         ranges::sort(g, {}, [&](P<T> p) { return pair(p.x, p.y); });
57         vector<P<T>> h;
58         for (auto p : g) {
59             while (ssize(h) >= 2 and
60                 sign((h.back() - h.end()[-2]).cross(p - h.back())) <= 0) {
61                 h.pop_back();
62             }
63             h.push_back(p);
64         }
65         int m = h.size();
66         for (auto p : g | views::reverse) {
67             while (ssize(h) > m and
68                 sign((h.back() - h.end()[-2]).cross(p - h.back())) <= 0) {
69                 h.pop_back();
70             }
71             h.push_back(p);
72         }
73         h.pop_back();
74         return G(h);
75     }
76     // Following function are valid only for convex.
77     T diameter2() {
78         int n = g.size();
79         T res = 0;
80         for (int i = 0, j = 1; i < n; i += 1) {
81             auto a = g[i], b = g[(i + 1) % n];
```

```

82     while (sign((b - a).cross(g[(j + 1) % n] - g[j])) > 0) {
83         j = (j + 1) % n;
84     }
85     res = max(res, (a - g[j]).len2());
86     res = max(res, (a - g[j]).len2());
87 }
88 return res;
89 }
90 optional<bool> contains(P<T> p) {
91     if (g[0] == p) { return {}; }
92     if (g.size() == 1) { return false; }
93     if (L(g[0], g[1]).contains(p)) { return {}; }
94     if (L(g[0], g[1]).left(p) <= 0) { return false; }
95     if (L(g[0], g.back()).left(p) > 0) { return false; }
96     int i = *ranges::partition_point(views::iota(2, ssize(g)), [&](int i) {
97         return sign((p - g[0]).cross(g[i] - g[0])) <= 0;
98     });
99     int s = L(g[i - 1], g[i]).left(p);
100     if (s == 0) { return {}; }
101     return s > 0;
102 }
103 int most(const function<P<T>(P<T>>>& f) {
104     int n = g.size();
105     auto check = [&](int i) {
106         return sign(f(g[i]).cross(g[(i + 1) % n] - g[i])) >= 0;
107     };
108     P<T> f0 = f(g[0]);
109     bool check0 = check(0);
110     if (not check0 and check(n - 1)) { return 0; }
111     return *ranges::partition_point(views::iota(0, n), [&](int i) -> bool {
112         if (i == 0) { return true; }
113         bool checki = check(i);
114         int t = sign(f0.cross(g[i] - g[0]));
115         if (i == 1 and checki == check0 and t == 0) { return true; }
116         return checki ^ (checki == check0 and t <= 0);
117     });
118 }
119 pair<int, int> tan(P<T> p) {
120     return {most([&](P<T> x) { return x - p; }},
121            most([&](P<T> x) { return p - x; }));}
122 }
123 pair<int, int> tan(L<T> l) {
124     return {most([&](P<T> _) { return l.v(); }},
125            most([&](P<T> _) { return -l.v(); }));}

```

```

126     }
127 };
128
129 template <typename T> vector<L<T>> half(vector<L<T>> ls, T bound) {
130     // Ranges: bound ~ 6
131     auto check = [&](L<T> a, L<T> b, L<T> c) {
132         auto [x, y] = b.intersection(c).value();
133         a = L(a.a * y, a.b * y);
134         return a.left(b.a * y + b.v() * x) < 0;
135     };
136     ls.emplace_back(P(-bound, (T)0), P(-bound, -(T)1));
137     ls.emplace_back(P((T)0, -bound), P((T)1, -bound));
138     ls.emplace_back(P(bound, (T)0), P(bound, (T)1));
139     ls.emplace_back(P((T)0, bound), P(-(T)1, bound));
140     ranges::sort(ls, [&](L<T> lhs, L<T> rhs) {
141         if (sign(lhs.v().cross(rhs.v())) == 0 and
142             sign(lhs.v().dot(rhs.v())) >= 0) {
143             return lhs.left(rhs.a) == -1;
144         }
145         return argument(lhs.v(), rhs.v());
146     });
147     deque<L<T>> q;
148     for (int i = 0; i < ssize(ls); i += 1) {
149         if (i and sign(ls[i - 1].v().cross(ls[i].v())) == 0 and
150             sign(ls[i - 1].v().dot(ls[i].v())) == 1) {
151             continue;
152         }
153         while (q.size() > 1 and check(ls[i], q.back(), q.end()[-2])) {
154             q.pop_back();
155         }
156         while (q.size() > 1 and check(ls[i], q[0], q[1])) { q.pop_front(); }
157         if (not q.empty() and sign(q.back().v().cross(ls[i].v())) <= 0) {
158             return {};
159         }
160         q.push_back(ls[i]);
161     }
162     while (q.size() > 1 and check(q[0], q.back(), q.end()[-2])) {
163         q.pop_back();
164     }
165     while (q.size() > 1 and check(q.back(), q[0], q[1])) { q.pop_front(); }
166     return vector<L<T>>(q.begin(), q.end());
167 }

```