

Team Reference Document

Heltion

March 8, 2024

Contents

1	Contest	1
1.1	Makefile	1
1.2	debug.h	1
1.3	Template	1
2	Graph	1
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	6
2.7	Minimum Perfect Matching on Bipartite Graph	6
2.8	Matching on General Graph	7
2.9	Maximum Flow	8
2.10	Minimum Cost Maximum Flow	9
3	Data Structure	10
3.1	Disjoint Set Union	10
3.2	Sparse Table	11
3.3	Treap	11
3.4	Lines Maximum	11
3.5	Segments Maximum	12
3.6	Segment Beats	13
3.7	Tree	14
3.7.1	Least Common Ancestor	14
3.7.2	Link Cut Tree	14
4	String	15
4.1	Z	15
4.2	Lyndon Factorization	16
4.3	Border	16
4.4	Manacher	16
4.5	Suffix Array	16
4.6	Aho-Corasick Automaton	17
4.7	Suffix Automaton	17
4.8	Palindromic Tree	18
5	Number Theory	18
5.1	Gaussian Integer	18
5.2	Modular Arithmetic	18
5.2.1	Sqrt	18
5.2.2	Logarithm	19
5.3	Chinese Remainder Theorem	19
5.4	Miller Rabin	19
5.5	Pollard Rho	20
5.6	Primitive Root	20
5.7	Sum of Floor	20
5.8	Minimum of Remainder	21
5.9	Stern Brocot Tree	21
5.10	Nim Product	21
6	Numerical	22
6.1	Golden Search	22
6.2	Adaptive Simpson	22
6.3	Simplex	22
6.4	Green’s Theorem	23
6.5	Double Integral	23
7	Convolution	23
7.1	Fast Fourier Transform on \mathbb{C}	23
7.2	Formal Power Series on \mathbb{F}_p	23
7.2.1	Newton’s Method	24
7.2.2	Arithmetic	24
7.2.3	Interpolation	24
7.2.4	Primes with root 3	24
7.3	Circular Transform	24
7.4	Truncated Transform	24
8	Geometry	24
8.1	Pick’s Theorem	24
8.2	2D Geometry	24

1 Contest

1.1 Makefile

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

1.2 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<ostreamstream>(ostreamstream() << 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"
```

1.3 Template

```
1 #include <bits/extc++.h>
2 using namespace std;
3 using namespace __gnu_pbds;
4 #ifndef ONLINE_JUDGE
5 #include "debug.h"
6 #else
7 #define debug(...) void(0)
8 #endif
```

```
9 template <typename T>
10 using RBTtree = tree<T,
11                     null_type,
12                     less<T>,
13                     rb_tree_tag,
14                     tree_order_statistics_node_update>;
15 using i64 = int64_t;
16 int main() {
17     cin.tie(nullptr)->sync_with_stdio(false);
18     cout << fixed << setprecision(20);
19 }
```

2 Graph

2.1 Connected Components

2.1.1 Strongly Connected Components

Returns strongly connected components in topologically order.

```
1 vector<vector<int>> strongly_connected_components(
2     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]) {
12                low = min(low, ~pos[v] ? pos[v] : dfs(v));
13            }
14        }
15        if (low == pos[u]) {
16            res.emplace_back(stack.begin() + low, stack.end());
17            for (int v : res.back()) {
18                done[v] = true;
19            }
20            stack.resize(low);
21        }
22        return low;
23    };
24    for (int i = 0; i < n; i += 1) {
25        if (not done[i]) {
26            dfs(i);
27        }
28    }
29    ranges::reverse(res);
30    return res;
31 }
```

2.1.2 Two-vertex-connected Components

```
1 vector<vector<int>> two_vertex_connected_components(  
2     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++) {  
11                     low = min(low, pos[v]);  
12                 }  
13             } else {  
14                 low = min(low, dfs(v, u));
```

```
15     }  
16 }  
17 if (low == pos[u]) {  
18     res.emplace_back(stack.begin() + low, stack.end());  
19     stack.resize(low);  
20 }  
21 return low;  
22 };  
23 for (int i = 0; i < n; i += 1) {  
24     if (pos[i] == -1) {  
25         dfs(i, -1);  
26     }  
27 }  
28 return res;  
29 }
```

2.1.4 Three-edge-connected Components

```
1 vector<vector<int>> three_edge_connected_components(  
2     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                 } 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) {  
31                             v = path[v];  
32                         }  
33                         if (low[u] > low[v]) {  
34                             low[u] = min(low[u], low[v]);
```

```

35         swap(v, path[u]);
36     }
37     for (; v != -1; v = path[v]) {
38         dsu.merge(u, v);
39         deg[u] += deg[v];
40     }
41 }
42 }
43 }
44 }
45 post[u] = dft;
46 };
47 for (int i = 0; i < n; i += 1) {
48     if (pre[i] == -1) {
49         dfs(i, -1);
50     }
51 }
52 vector<vector<int>> _res(n);
53 for (int i = 0; i < n; i += 1) {
54     _res[dsu.find(i)].push_back(i);
55 }
56 vector<vector<int>> res;
57 for (auto& res_i : _res) {
58     if (not res_i.empty()) {
59         res.emplace_back(move(res_i));
60     }
61 }
62 return res;
63 }

```

2.2 Euler Walks

```

1 optional<vector<vector<pair<int, bool>>>> undirected_walks(
2     int n,
3     const vector<pair<int, int>>& edges) {
4     int m = ssize(edges);
5     vector<vector<pair<int, bool>>> res;
6     if (not m) {
7         return res;
8     }
9     vector<vector<pair<int, bool>>> g(n);
10    for (int i = 0; i < m; i += 1) {
11        auto [u, v] = edges[i];
12        g[u].emplace_back(i, true);
13        g[v].emplace_back(i, false);
14    }
15    for (int i = 0; i < n; i += 1) {
16        if (g[i].size() % 2) {
17            return {};
18        }
19    }
20    vector<pair<int, bool>> walk;

```

```

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

```

```

74     } else {
75         i += 1;
76     }
77 }
78 };
79 for (int i = 0; i < n; i += 1) {
80     dfs(i);
81     if (not walk.empty()) {
82         ranges::reverse(walk);
83         res.emplace_back(move(walk));
84     }
85 }
86 return res;
87 }

```

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), sdom(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] = sdom[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]) {
19             return x ? -1 : u;
20         }
21         int v = find(dsu[u], x + 1);
22         if (v < 0) {
23             return u;
24         }
25         if (sdom[label[dsu[u]]] < sdom[label[u]]) {
26             label[u] = label[dsu[u]];
27         }
28         dsu[u] = v;
29         return x ? v : label[u];
30     };
31     dfs(s);
32     iota(dom.begin(), dom.end(), 0);
33     for (int i = ssize(p) - 1; i >= 0; i -= 1) {
34         for (int j : rg[i]) {
35             sdom[i] = min(sdom[i], sdom[find(j, 0)]);

```

```

36     }
37     if (i) {
38         bucket[sdom[i]].push_back(i);
39     }
40     for (int k : bucket[i]) {
41         int j = find(k, 0);
42         dom[k] = sdom[j] == sdom[k] ? sdom[j] : j;
43     }
44     if (i > 1) {
45         dsu[i] = par[i];
46     }
47 }
48 for (int i = 1; i < ssize(p); i += 1) {
49     if (dom[i] != sdom[i]) {
50         dom[i] = dom[dom[i]];
51     }
52 }
53 vector<int> res(n, -1);
54 res[s] = s;
55 for (int i = 1; i < ssize(p); i += 1) {
56     res[p[i]] = p[dom[i]];
57 }
58 return res;
59 }

```

2.4 Directed Minimum Spanning Tree

```

1 struct Node {
2     Edge e;
3     int d;
4     Node *l, *r;
5     Node(Edge e)
6         : e(e), d(0) { l = r = nullptr; }
7     void add(int v) {
8         e.w += v;
9         d += v;
10    }
11    void push() {
12        if (l) {
13            l->add(d);
14        }
15        if (r) {
16            r->add(d);
17        }
18        d = 0;
19    }
20 };
21 Node* merge(Node* u, Node* v) {
22     if (not u or not v) {
23         return u ? v : u;
24     }
25     if (u->e.w > v->e.w) {

```

```

26     swap(u, v);
27 }
28 u->push();
29 u->r = merge(u->r, v);
30 swap(u->l, u->r);
31 return u;
32 }
33 void pop(Node*& u) {
34     u->push();
35     u = merge(u->l, u->r);
36 }
37 pair<i64, vector<int>>
38 directed_minimum_spanning_tree(int n, const vector<Edge>& edges, int s) {
39     i64 ans = 0;
40     vector<Node*> heap(n), edge(n);
41     RollbackDisjointSetUnion dsu(n), rbdsu(n);
42     vector<pair<Node*, int>> cycles;
43     for (auto e : edges) {
44         heap[e.v] = merge(heap[e.v], new Node(e));
45     }
46     for (int i = 0; i < n; i += 1) {
47         if (i == s) {
48             continue;
49         }
50         for (int u = i;;) {
51             if (not heap[u]) {
52                 return {};
53             }
54             ans += (edge[u] = heap[u])->e.w;
55             edge[u]->add(-edge[u]->e.w);
56             int v = rbdsu.find(edge[u]->e.u);
57             if (dsu.merge(u, v)) {
58                 break;
59             }
60             int t = rbdsu.time();
61             while (rbdsu.merge(u, v)) {
62                 heap[rbdsu.find(u)] = merge(heap[u], heap[v]);
63                 u = rbdsu.find(u);
64                 v = rbdsu.find(edge[v]->e.u);
65             }
66             cycles.emplace_back(edge[u], t);
67             while (heap[u] and rbdsu.find(heap[u]->e.u) == rbdsu.find(u)) {
68                 pop(heap[u]);
69             }
70         }
71     }
72     for (auto [p, t] : cycles | views::reverse) {
73         int u = rbdsu.find(p->e.v);
74         rbdsu.rollback(t);
75         int v = rbdsu.find(edge[u]->e.v);
76         edge[v] = exchange(edge[u], p);
77     }
78     vector<int> res(n, -1);

```

```

79     for (int i = 0; i < n; i += 1) {
80         res[i] = i == s ? i : edge[i]->e.u;
81     }
82     return {ans, res};
83 }

```

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)
6         : v(v), w(w), h(1) { l = r = nullptr; }
7 };
8 Node* merge(Node* u, Node* v) {
9     if (not u or not v) {
10         return u ?: v;
11     }
12     if (u->w > v->w) {
13         swap(u, v);
14     }
15     Node* p = new Node(*u);
16     p->r = merge(u->r, v);
17     if (p->r and (not p->l or p->l->h < p->r->h)) {
18         swap(p->l, p->r);
19     }
20     p->h = (p->r ? p->r->h : 0) + 1;
21     return p;
22 }
23 struct Edge {
24     int u, v, w;
25 };
26 template <typename T>
27 using minimum_heap = priority_queue<T, vector<T>, greater<T>>;
28 vector<i64> k_shortest_paths(int n,
29                             const vector<Edge>& edges,
30                             int s,
31                             int t,
32                             int k) {
33     vector<vector<int>> g(n);
34     for (int i = 0; i < ssize(edges); i += 1) {
35         g[edges[i].u].push_back(i);
36     }
37     vector<int> par(n, -1), p;
38     vector<i64> d(n, -1);
39     minimum_heap<pair<i64, int>> pq;
40     pq.push({d[s] = 0, s});
41     while (not pq.empty()) {
42         auto [du, u] = pq.top();
43         pq.pop();
44         if (du > d[u]) {

```

```

45     continue;
46 }
47 p.push_back(u);
48 for (int i : g[u]) {
49     auto [_, v, w] = edges[i];
50     if (d[v] == -1 or d[v] > d[u] + w) {
51         par[v] = i;
52         pq.push({d[v] = d[u] + w, v});
53     }
54 }
55 }
56 if (d[t] == -1) {
57     return vector<i64>(k, -1);
58 }
59 vector<Node*> heap(n);
60 for (int i = 0; i < ssize(edges); i += 1) {
61     auto [u, v, w] = edges[i];
62     if (~d[u] and ~d[v] and par[v] != i) {
63         heap[v] = merge(heap[v], new Node(u, d[u] + w - d[v]));
64     }
65 }
66 for (int u : p) {
67     if (u != s) {
68         heap[u] = merge(heap[u], heap[edges[par[u]].u]);
69     }
70 }
71 minimum_heap<pair<i64, Node*>> q;
72 if (heap[t]) {
73     q.push({d[t] + heap[t]->w, heap[t]});
74 }
75 vector<i64> res = {d[t]};
76 for (int i = 1; i < k and not q.empty(); i += 1) {
77     auto [w, p] = q.top();
78     q.pop();
79     res.push_back(w);
80     if (heap[p->v]) {
81         q.push({w + heap[p->v]->w, heap[p->v]});
82     }
83     for (auto c : {p->l, p->r}) {
84         if (c) {
85             q.push({w + c->w - p->w, c});
86         }
87     }
88 }
89 res.resize(k, -1);
90 return res;
91 }

```

2.6 Global Minimum Cut

```

1 i64 global_minimum_cut(vector<vector<i64>>& w) {
2     int n = w.size();

```

```

3     if (n == 2) {
4         return w[0][1];
5     }
6     vector<bool> in(n);
7     vector<int> add;
8     vector<i64> s(n);
9     i64 st = 0;
10    for (int i = 0; i < n; i += 1) {
11        int k = -1;
12        for (int j = 0; j < n; j += 1) {
13            if (not in[j]) {
14                if (k == -1 or s[j] > s[k]) {
15                    k = j;
16                }
17            }
18        }
19        add.push_back(k);
20        st = s[k];
21        in[k] = true;
22        for (int j = 0; j < n; j += 1) {
23            s[j] += w[j][k];
24        }
25    }
26    for (int i = 0; i < n; i += 1) {
27    }
28    int x = add.rbegin()[1], y = add.back();
29    if (x == n - 1) {
30        swap(x, y);
31    }
32    for (int i = 0; i < n; i += 1) {
33        swap(w[y][i], w[n - 1][i]);
34        swap(w[i][y], w[i][n - 1]);
35    }
36    for (int i = 0; i + 1 < n; i += 1) {
37        w[i][x] += w[i][n - 1];
38        w[x][i] += w[n - 1][i];
39    }
40    w.pop_back();
41    return min(st, stoer_wagner(w));
42 }

```

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) {
19         continue;
20     }
21     vector<i64> d(n);
22     for (int c = 0; c < n; c += 1) {
23         d[c] = resid(r, c);
24     }
25     vector<int> pre(n, r);
26     int scan = 0, label = 0, last = 0, col = -1;
27     [&]() {
28         while (true) {
29             if (scan == label) {
30                 last = scan;
31                 i64 min = d[cols[scan]];
32                 for (int j = scan; j < n; j += 1) {
33                     int c = cols[j];
34                     if (d[c] <= min) {
35                         if (d[c] < min) {
36                             min = d[c];
37                             label = scan;
38                         }
39                         swap(cols[j], cols[label++]);
40                     }
41                 }
42                 for (int j = scan; j < label; j += 1) {
43                     if (int c = cols[j]; cm[c] == -1) {
44                         col = c;
45                         return;
46                     }
47                 }
48             }
49             int c1 = cols[scan++], r1 = cm[c1];
50             for (int j = label; j < n; j += 1) {
51                 int c2 = cols[j];
52                 i64 len = resid(r1, c2) - resid(r1, c1);
53                 if (d[c2] > d[c1] + len) {
54                     d[c2] = d[c1] + len;
55                     pre[c2] = r1;
56                     if (len == 0) {
57                         if (cm[c2] == -1) {
58                             col = c2;
59                             return;
60                         }
61                         swap(cols[j], cols[label++]);
62                     }

```

```

63         }
64     }
65 }
66 }();
67 for (int i = 0; i < last; i += 1) {
68     int c = cols[i];
69     pi[c] += d[c] - d[col];
70 }
71 for (int t = col; t != -1;) {
72     col = t;
73     int r = pre[col];
74     cm[col] = r;
75     swap(rm[r], t);
76 }
77 }
78 i64 res = 0;
79 for (int i = 0; i < n; i += 1) {
80     res += w[i][rm[i]];
81 }
82 return {res, rm};
83 }

```

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) {
37                     return u;
38                 }
39                 book[u] = mark;
40                 u = c[par[matched[u]]];
41             }
42             swap(u, v);
43         }
44     }(u, v);
45     auto up = [&](int u, int v, int w) {
46         while (c[u] != w) {
47             par[u] = v;
48             v = matched[u];
49             if (type[v] == 1) {
50                 q.push(v);
51                 type[v] == 0;
52             }
53             if (c[u] == u) {
54                 c[u] = w;
55             }
56             if (c[v] == v) {
57                 c[v] = w;
58             }
59             u = par[v];
60         }
61     };
62     up(u, v, w);
63     up(v, u, w);
64     for (int i = 0; i < n; i += 1) {
65         c[i] = c[c[i]];
66     }
67 }
68 }
69 };
70 for (int i = 0; i < n; i += 1) {
71     if (matched[i] == -1) {
72         match(i);
73     }
74 }
75 return matched;
76 }

```

2.9 Maximum Flow

```

1 struct HighestLabelPreflowPush {

```

```

2     int n;
3     vector<vector<int>> g;
4     vector<Edge> edges;
5     HighestLabelPreflowPush(int n)
6         : n(n), g(n) {}
7     int add(int u, int v, i64 f) {
8         if (u == v) {
9             return -1;
10        }
11        int i = ssize(edges);
12        edges.push_back({u, v, f});
13        g[u].push_back(i);
14        edges.push_back({v, u, 0});
15        g[v].push_back(i + 1);
16        return i;
17    }
18    i64 max_flow(int s, int t) {
19        vector<i64> p(n);
20        vector<int> h(n), cur(n), count(n * 2);
21        vector<vector<int>> pq(n * 2);
22        auto push = [&](int i, i64 f) {
23            auto [u, v, _] = edges[i];
24            if (not p[v] and f) {
25                pq[h[v]].push_back(v);
26            }
27            edges[i].f -= f;
28            edges[i ^ 1].f += f;
29            p[u] -= f;
30            p[v] += f;
31        };
32        h[s] = n;
33        count[0] = n - 1;
34        p[t] = 1;
35        for (int i : g[s]) {
36            push(i, edges[i].f);
37        }
38        for (int hi = 0;;) {
39            while (pq[hi].empty()) {
40                if (not hi--) {
41                    return -p[s];
42                }
43            }
44            int u = pq[hi].back();
45            pq[hi].pop_back();
46            while (p[u] > 0) {
47                if (cur[u] == ssize(g[u])) {
48                    h[u] = n * 2 + 1;
49                    for (int i = 0; i < ssize(g[u]); i += 1) {
50                        auto [_, v, f] = edges[g[u][i]];
51                        if (f and h[u] > h[v] + 1) {
52                            h[u] = h[v] + 1;
53                            cur[u] = i;
54                        }

```

```

55     }
56     count[h[u]] += 1;
57     if (not(count[hi] == 1) and hi < n) {
58         for (int i = 0; i < n; i += 1) {
59             if (h[i] > hi and h[i] < n) {
60                 count[h[i]] -= 1;
61                 h[i] = n + 1;
62             }
63         }
64     }
65     hi = h[u];
66 } else {
67     int i = g[u][cur[u]];
68     auto [_, v, f] = edges[i];
69     if (f and h[u] == h[v] + 1) {
70         push(i, min(p[u], f));
71     } else {
72         cur[u] += 1;
73     }
74 }
75 }
76 }
77 return i64(0);
78 }
79 };
80
81 struct Dinic {
82     int n;
83     vector<vector<int>> g;
84     vector<Edge> edges;
85     vector<int> level;
86     Dinic(int n)
87         : n(n), g(n) {}
88     int add(int u, int v, i64 f) {
89         if (u == v) {
90             return -1;
91         }
92         int i = ssize(edges);
93         edges.push_back({u, v, f});
94         g[u].push_back(i);
95         edges.push_back({v, u, 0});
96         g[v].push_back(i + 1);
97         return i;
98     }
99     i64 max_flow(int s, int t) {
100         i64 flow = 0;
101         queue<int> q;
102         vector<int> cur;
103         auto bfs = [&]() {
104             level.assign(n, -1);
105             level[s] = 0;
106             q.push(s);
107             while (not q.empty()) {

```

```

108                 int u = q.front();
109                 q.pop();
110                 for (int i : g[u]) {
111                     auto [_, v, c] = edges[i];
112                     if (c and level[v] == -1) {
113                         level[v] = level[u] + 1;
114                         q.push(v);
115                     }
116                 }
117             }
118             return ~level[t];
119         };
120         auto dfs = [&](auto& dfs, int u, i64 limit) -> i64 {
121             if (u == t) {
122                 return limit;
123             }
124             i64 res = 0;
125             for (int& i = cur[u]; i < ssize(g[u]) and limit; i += 1) {
126                 int j = g[u][i];
127                 auto [_, v, f] = edges[j];
128                 if (level[v] == level[u] + 1 and f) {
129                     if (i64 d = dfs(dfs, v, min(f, limit)); d) {
130                         limit -= d;
131                         res += d;
132                         edges[j].f -= d;
133                         edges[j ^ 1].f += d;
134                     }
135                 }
136             }
137             return res;
138         };
139         while (bfs()) {
140             cur.assign(n, 0);
141             while (i64 f = dfs(dfs, s, numeric_limits<i64>::max())) {
142                 flow += f;
143             }
144         }
145         return flow;
146     }
147 };

```

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)
8         : n(n), g(n) {}

```

```

9  int add_edge(int u, int v, i64 f, i64 c) {
10     int i = edges.size();
11     edges.push_back({u, v, f, c});
12     edges.push_back({v, u, 0, -c});
13     g[u].push_back(i);
14     g[v].push_back(i + 1);
15     return i;
16 }
17 pair<i64, i64> flow(int s, int t) {
18     constexpr i64 inf = numeric_limits<i64>::max();
19     vector<i64> d, h(n);
20     vector<int> p;
21     auto dijkstra = [&]() {
22         d.assign(n, inf);
23         p.assign(n, -1);
24         minimum_heap<pair<i64, int>> q;
25         q.emplace(d[s] = 0, s);
26         while (not q.empty()) {
27             auto [du, u] = q.top();
28             q.pop();
29             if (du > d[u]) {
30                 continue;
31             }
32             for (int i : g[u]) {
33                 auto [_, v, f, c] = edges[i];
34                 if (f and d[v] > d[u] + h[u] - h[v] + c) {
35                     p[v] = i;
36                     q.emplace(d[v] = d[u] + h[u] - h[v] + c, v);
37                 }
38             }
39         }
40         return ~p[t];
41     };
42     i64 f = 0, c = 0;
43     while (dijkstra()) {
44         for (int i = 0; i < n; i += 1) {
45             h[i] += d[i];
46         }
47         vector<int> path;
48         for (int u = t; u != s; u = edges[p[u]].u) {
49             path.push_back(p[u]);
50         }
51         i64 mf =
52             edges[ranges::min(path, {}, [&](int i) { return edges[i].f; } )].f;
53         f += mf;
54         c += mf * h[t];
55         for (int i : path) {
56             edges[i].f -= mf;
57             edges[i ^ 1].f += mf;
58         }
59     }
60     return {f, c};
61 }

```

```
62 };
```

3 Data Structure

3.1 Disjoint Set Union

```

1  struct DisjointSetUnion {
2      vector<int> dsu;
3      DisjointSetUnion(int n)
4          : dsu(n, -1) {}
5      int find(int u) { return dsu[u] < 0 ? u : dsu[u] = find(dsu[u]); }
6      void merge(int u, int v) {
7          u = find(u);
8          v = find(v);
9          if (u != v) {
10             if (dsu[u] > dsu[v]) {
11                 swap(u, v);
12             }
13             dsu[u] += dsu[v];
14             dsu[v] = u;
15         }
16     }
17 };
18 struct RollbackDisjointSetUnion {
19     vector<pair<int, int>> stack;
20     vector<int> dsu;
21     RollbackDisjointSetUnion(int n)
22         : dsu(n, -1) {}
23     int find(int u) { return dsu[u] < 0 ? u : find(dsu[u]); }
24     int time() { return ssize(stack); }
25     bool merge(int u, int v) {
26         if ((u = find(u)) == (v = find(v))) {
27             return false;
28         }
29         if (dsu[u] < dsu[v]) {
30             swap(u, v);
31         }
32         stack.emplace_back(u, dsu[u]);
33         dsu[v] += dsu[u];
34         dsu[u] = v;
35         return true;
36     }
37     void rollback(int t) {
38         while (ssize(stack) > t) {
39             auto [u, dsu_u] = stack.back();
40             stack.pop_back();
41             dsu[dsu[u]] -= dsu_u;
42             dsu[u] = dsu_u;
43         }
44     }
45 };

```

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 - l)) - 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) {
38            return table[0][l];
39        }
40        int i = bit_width(unsigned(l ^ (r - 1))) - 1;
41        return min(table[i][l], table[i][r - 1]);
42    }
43 };
```

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)
10        : v(v), sum(v), priority(mt()), size(1) { l = r = nullptr; }
11    Node* update(Node* l, Node* r) {
12        Node* p = persistent ? new Node(*this) : this;
13        p->l = l;
14        p->r = r;
15        p->size = (l ? l->size : 0) + 1 + (r ? r->size : 0);
16        p->sum = (l ? l->sum : 0) + v + (r ? r->sum : 0);
17        return p;
18    }
19 };
20 mt19937_64 Node::mt;
21 pair<Node*, Node*> split_by_v(Node* p, int v) {
22     if (not p) {
23         return {};
24     }
25     if (p->v < v) {
26         auto [l, r] = split_by_v(p->r, v);
27         return {p->update(p->l, l), r};
28     }
29     auto [l, r] = split_by_v(p->l, v);
30     return {l, p->update(r, p->r)};
31 }
32 pair<Node*, Node*> split_by_size(Node* p, int size) {
33     if (not p) {
34         return {};
35     }
36     int l_size = p->l ? p->l->size : 0;
37     if (l_size < size) {
38         auto [l, r] = split_by_size(p->r, size - l_size - 1);
39         return {p->update(p->l, l), r};
40     }
41     auto [l, r] = split_by_size(p->l, size);
42     return {l, p->update(r, p->r)};
43 }
44 Node* merge(Node* l, Node* r) {
45     if (not l or not r) {
46         return l ? r;
47     }
48     if (l->priority < r->priority) {
49         return r->update(merge(l, r->l), r->r);
50     }
51     return l->update(l->l, merge(l->r, r));
52 }
```

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()) {
11             return x->p == inf, false;
12         }
13         if (x->k == y->k) {
14             x->p = x->b > y->b ? inf : -inf;
15         } else {
16             x->p = div(y->b - x->b, x->k - y->k);
17         }
18         return x->p >= y->p;
19     }
20     void add(i64 k, i64 b) {
21         auto z = insert({k, b, 0}), y = z++, x = y;
22         while (isect(y, z)) {
23             z = erase(z);
24         }
25         if (x != begin() and isect(--x, y)) {
26             isect(x, y = erase(y));
27         }
28         while ((y = x) != begin() and (--x)->p >= y->p) {
29             isect(x, erase(y));
30         }
31     }
32     optional<i64> get(i64 x) {
33         if (empty()) {
34             return {};
35         }
36         auto it = lower_bound(x);
37         return it->k * x + it->b;
38     }
39 };

```

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)
13     : tl(tl), tr(tr), root(nullptr) {}
14 void add(i64 l, i64 r, i64 k, i64 b) {
15     function<void(Node*&, i64, i64, Segment)> rec = [&](Node*& p, i64 tl,
16                                                         i64 tr, Segment s) {
17         if (p == nullptr) {
18             p = new Node();
19         }
20         i64 tm = midpoint(tl, tr);
21         if (tl >= l and tr <= r) {
22             if (not p->s) {
23                 p->s = s;
24                 return;
25             }
26             auto t = p->s.value();
27             if (t.get(tl) >= s.get(tl)) {
28                 if (t.get(tr) >= s.get(tr)) {
29                     return;
30                 }
31                 if (t.get(tm) >= s.get(tm)) {
32                     return rec(p->r, tm + 1, tr, s);
33                 }
34                 p->s = s;
35                 return rec(p->l, tl, tm, t);
36             }
37             if (t.get(tr) <= s.get(tr)) {
38                 p->s = s;
39                 return;
40             }
41             if (t.get(tm) <= s.get(tm)) {
42                 p->s = s;
43                 return rec(p->r, tm + 1, tr, t);
44             }
45             return rec(p->l, tl, tm, s);
46         }
47         if (l <= tm) {
48             rec(p->l, tl, tm, s);
49         }
50         if (r > tm) {
51             rec(p->r, tm + 1, tr, s);
52         }
53     };
54     rec(root, tl, tr, {k, b});
55 }
56 optional<i64> get(i64 x) {
57     optional<i64> res = {};
58     function<void(Node*, i64, i64)> rec = [&](Node* p, i64 tl, i64 tr) {
59         if (p == nullptr) {
60             return;
61         }
62         i64 tm = midpoint(tl, tr);
63         if (p->s) {

```

```

64     i64 y = p->s.value().get(x);
65     if (not res or res.value() < y) {
66         res = y;
67     }
68 }
69 if (x <= tm) {
70     rec(p->l, tl, tm);
71 } else {
72     rec(p->r, tm + 1, tr);
73 }
74 };
75 rec(root, tl, tr);
76 return res;
77 }
78 };

```

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)
8         : less(less) {
9         mv = x;
10        smv = tmv = def();
11        cmv = 1;
12    }
13    void up(const Mv& ls, const Mv& rs) {
14        mv = mmv(ls.mv, rs.mv);
15        smv = mmv(ls.mv == mv ? ls.smv : ls.mv, rs.mv == mv ? rs.smv : rs.mv);
16        cmv = (ls.mv == mv ? ls.cmv : 0) + (rs.mv == mv ? rs.cmv : 0);
17    }
18    void add(i64 x) {
19        mv += x;
20        if (smv != def()) {
21            smv += x;
22        }
23        if (tmv != def()) {
24            tmv += x;
25        }
26    }
27 };
28 struct Node {
29     Mv mn, mx;
30     i64 sum, tsum;
31     Node *ls, *rs;
32     Node(i64 x = 0)
33         : sum(x), tsum(0), mn(x, true), mx(x, false) {
34         ls = rs = nullptr;

```

```

35     }
36     void up() {
37         sum = ls->sum + rs->sum;
38         mx.up(ls->mx, rs->mx);
39         mn.up(ls->mn, rs->mn);
40     }
41     void down(int tl, int tr) {
42         if (tsum) {
43             int tm = midpoint(tl, tr);
44             ls->add(tl, tm, tsum);
45             rs->add(tm, tr, tsum);
46             tsum = 0;
47         }
48         if (mn.tmv != mn.def()) {
49             ls->ch(mn.tmv, true);
50             rs->ch(mn.tmv, true);
51             mn.tmv = mn.def();
52         }
53         if (mx.tmv != mx.def()) {
54             ls->ch(mx.tmv, false);
55             rs->ch(mx.tmv, false);
56             mx.tmv = mx.def();
57         }
58     }
59     bool cmp(i64 x, i64 y, bool less) { return less ? x < y : x > y; }
60     void add(int tl, int tr, i64 x) {
61         sum += (tr - tl) * x;
62         tsum += x;
63         mx.add(x);
64         mn.add(x);
65     }
66     void ch(i64 x, bool less) {
67         auto &lhs = less ? mn : mx, &rhs = less ? mx : mn;
68         if (not cmp(x, rhs.mv, less)) {
69             return;
70         }
71         sum += (x - rhs.mv) * rhs.cmv;
72         if (lhs.smv == rhs.mv) {
73             lhs.smv = x;
74         }
75         if (lhs.mv == rhs.mv) {
76             lhs.mv = x;
77         }
78         if (cmp(x, rhs.tmv, less)) {
79             rhs.tmv = x;
80         }
81         rhs.mv = lhs.tmv = x;
82     }
83     void add(int tl, int tr, int l, int r, i64 x) {
84         if (tl >= l and tr <= r) {
85             return add(tl, tr, x);
86         }
87         down(tl, tr);

```

```

88     int tm = midpoint(tl, tr);
89     if (l < tm) {
90         ls->add(tl, tm, l, r, x);
91     }
92     if (r > tm) {
93         rs->add(tm, tr, l, r, x);
94     }
95     up();
96 }
97 void ch(int tl, int tr, int l, int r, i64 x, bool less) {
98     auto &lhs = less ? mn : mx, &rhs = less ? mx : mn;
99     if (not cmp(x, rhs.mv, less)) {
100         return;
101     }
102     if (tl >= l and tr <= r and cmp(rhs.smv, x, less)) {
103         return ch(x, less);
104     }
105     down(tl, tr);
106     int tm = midpoint(tl, tr);
107     if (l < tm) {
108         ls->ch(tl, tm, l, r, x, less);
109     }
110     if (r > tm) {
111         rs->ch(tm, tr, l, r, x, less);
112     }
113     up();
114 }
115 i64 get(int tl, int tr, int l, int r) {
116     if (tl >= l and tr <= r) {
117         return sum;
118     }
119     down(tl, tr);
120     i64 res = 0;
121     int tm = midpoint(tl, tr);
122     if (l < tm) {
123         res += ls->get(tl, tm, l, r);
124     }
125     if (r > tm) {
126         res += rs->get(tm, tr, l, r);
127     }
128     return res;
129 }
130 };

```

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) {
21            a[i] = time[par[p[i]]];
22        }
23        st = SparseTable(a);
24    }
25    int query(int u, int v) {
26        if (u == v) {
27            return u;
28        }
29        if (time[u] > time[v]) {
30            swap(u, v);
31        }
32        return p[st.query(time[u] + 1, time[v] + 1)];
33    }
34 };

```

3.7.2 Link Cut Tree

```

1 template <class T, class E, class REV, class OP>
2 struct Node {
3     T t, st;
4     bool reversed;
5     Node* par;
6     array<Node*, 2> ch;
7     Node(T t = E()) {
8         : t(t), st(t), reversed(false), par(nullptr) {
9         ch.fill(nullptr);
10    }
11    int get_s() {
12        if (par == nullptr) {
13            return -1;
14        }
15        if (par->ch[0] == this) {
16            return 0;
17        }
18        if (par->ch[1] == this) {
19            return 1;

```



```

20     }
21     return -1;
22 }
23 void push_up() {
24     st = OP()(ch[0] ? ch[0]->st : E(), OP()(t, ch[1] ? ch[1]->st : E()));
25 }
26 void reverse() {
27     reversed ^= 1;
28     st = REV()(st);
29 }
30 void push_down() {
31     if (reversed) {
32         swap(ch[0], ch[1]);
33         if (ch[0]) {
34             ch[0]->reverse();
35         }
36         if (ch[1]) {
37             ch[1]->reverse();
38         }
39         reversed = false;
40     }
41 }
42 void attach(int s, Node* u) {
43     if ((ch[s] = u)) {
44         u->par = this;
45     }
46     push_up();
47 }
48 void rotate() {
49     auto p = par;
50     auto pp = p->par;
51     int s = get_s();
52     int ps = p->get_s();
53     p->attach(s, ch[s ^ 1]);
54     attach(s ^ 1, p);
55     if (~ps) {
56         pp->attach(ps, this);
57     }
58     par = pp;
59 }
60 void splay() {
61     push_down();
62     while (~get_s() and ~par->get_s()) {
63         par->par->push_down();
64         par->push_down();
65         push_down();
66         (get_s() == par->get_s() ? par : this)->rotate();
67         rotate();
68     }
69     if (~get_s()) {
70         par->push_down();
71         push_down();
72         rotate();

```

```

73     }
74 }
75 void access() {
76     splay();
77     attach(1, nullptr);
78     while (par != nullptr) {
79         auto p = par;
80         p->splay();
81         p->attach(1, this);
82         rotate();
83     }
84 }
85 void make_root() {
86     access();
87     reverse();
88     push_down();
89 }
90 void link(Node* u) {
91     u->make_root();
92     access();
93     attach(1, u);
94 }
95 void cut(Node* u) {
96     u->make_root();
97     access();
98     if (ch[0] == u) {
99         ch[0] = u->par = nullptr;
100         push_up();
101     }
102 }
103 void set(T t) {
104     access();
105     this->t = t;
106     push_up();
107 }
108 T query(Node* u) {
109     u->make_root();
110     access();
111     return st;
112 }
113 };

```

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]]) {
7         z[i] += 1;
8     }
9     if (i + z[i] > j + z[j]) {
10        j = i;
11    }
12 }
13 return z;
14 }

```

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; i++) {
4         int j = i + 1, k = i;
5         for (; j < n and s[k] <= s[j]; j++) {
6             k = s[k] < s[j] ? i : k + 1;
7         }
8         while (i <= k) {
9             res.push_back(i + j - k);
10        }
11    }
12    return res;
13 }

```

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++) {
5         int& j = res[i] = res[i - 1];
6         while (j and s[i] != s[j]) {
7             j = res[j - 1];
8         }
9         j += s[i] == s[j];
10    }
11    return res;
12 }

```

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++) {
5         if (j + p[j] > i) {

```

```

6         p[i] = min(p[j * 2 - i], j + p[j] - i);
7     }
8     while (i >= p[i] and i + p[i] < n and s[i - p[i]] == s[i + p[i]]) {
9         p[i] += 1;
10    }
11    if (i + p[i] > j + p[j]) {
12        j = i;
13    }
14 }
15 return p;
16 }

```

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++) {
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) {
14                q.push_back(i - m);
15            }
16        }
17        count.assign(k, 0);
18        for (int i : rank) {
19            count[i] += 1;
20        }
21        partial_sum(count.begin(), count.end(), count.begin());
22        for (int i = n - 1; i >= 0; i--) {
23            p[count[rank[q[i]]] - 1] = q[i];
24        }
25        auto previous = rank;
26        previous.resize(2 * n, -1);
27        k = 0;
28        for (int i = 0; i < n; i++) {
29            rank[p[i]] = i and previous[p[i]] == previous[p[i - 1]] and
30                previous[p[i] + m] == previous[p[i - 1] + m]
31                ? rank[p[i - 1]]
32                : k++;
33        }
34    }
35    vector<int> lcp(n);
36    k = 0;
37    for (int i = 0; i < n; i++) {
38        if (rank[i]) {

```

```

39     k = max(k - 1, 0);
40     int j = p[rank[i] - 1];
41     while (i + k < n and j + k < n and s[i + k] == s[j + k]) {
42         k += 1;
43     }
44     lcp[rank[i]] = k;
45 }
46 }
47 return {p, lcp};
48 }

```

4.6 Aho-Corasick Automaton

```

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

```

```

40     }
41 }
42 }
43 };

```

4.7 Suffix Automaton

```

1 struct Node {
2     int link, len;
3     array<int, sigma> next;
4     Node()
5         : link(-1), len(0) { next.fill(-1); }
6 };
7 struct SuffixAutomaton : vector<Node> {
8     SuffixAutomaton()
9         : vector<Node>(1) {}
10     int extend(int p, int c) {
11         if (~at(p).next[c]) {
12             // For online multiple strings.
13             int q = at(p).next[c];
14             if (at(p).len + 1 == at(q).len) {
15                 return q;
16             }
17             int clone = size();
18             push_back(at(q));
19             back().len = at(p).len + 1;
20             while (~p and at(p).next[c] == q) {
21                 at(p).next[c] = clone;
22                 p = at(p).link;
23             }
24             at(q).link = clone;
25             return clone;
26         }
27         int cur = size();
28         emplace_back();
29         back().len = at(p).len + 1;
30         while (~p and at(p).next[c] == -1) {
31             at(p).next[c] = cur;
32             p = at(p).link;
33         }
34         if (~p) {
35             int q = at(p).next[c];
36             if (at(p).len + 1 == at(q).len) {
37                 back().link = q;
38             } else {
39                 int clone = size();
40                 push_back(at(q));
41                 back().len = at(p).len + 1;
42                 while (~p and at(p).next[c] == q) {
43                     at(p).next[c] = clone;
44                     p = at(p).link;
45                 }

```

```

46         at(q).link = at(cur).link = clone;
47     }
48 } else {
49     back().link = 0;
50 }
51 return cur;
52 }
53 };

```

4.8 Palindromic Tree

```

1 struct Node {
2     int sum, len, link;
3     array<int, sigma> next;
4     Node(int len)
5         : len(len) {
6         sum = link = 0;
7         next.fill(0);
8     }
9 };
10 struct PalindromicTree : vector<Node> {
11     int last;
12     vector<int> s;
13     PalindromicTree()
14         : last(0) {
15         emplace_back(0);
16         emplace_back(-1);
17         at(0).link = 1;
18     }
19     int get_link(int u, int i) {
20         while (i < at(u).len + 1 or s[i - at(u).len - 1] != s[i])
21             u = at(u).link;
22         return u;
23     }
24     void extend(int i) {
25         int cur = get_link(last, i);
26         if (not at(cur).next[s[i]]) {
27             int now = size();
28             emplace_back(at(cur).len + 2);
29             back().link = at(get_link(at(cur).link, i)).next[s[i]];
30             back().sum = at(back().link).sum + 1;
31             at(cur).next[s[i]] = now;
32         }
33         last = at(cur).next[s[i]];
34     }
35 };

```

5 Number Theory

5.1 Gaussian Integer

```

1 i64 div_floor(i64 x, i64 y) {
2     return x / y - (x % y < 0);
3 }
4 i64 div_ceil(i64 x, i64 y) {
5     return x / y + (x % y > 0);
6 }
7 i64 div_round(i64 x, i64 y) {
8     return div_floor(2 * x + y, 2 * y);
9 }
10 struct Gauss {
11     i64 x, y;
12     i64 norm() { return x * x + y * y; }
13     bool operator!=(i64 r) { return y or x != r; }
14     Gauss operator~() { return {x, -y}; }
15     Gauss operator-(Gauss rhs) { return {x - rhs.x, y - rhs.y}; }
16     Gauss operator*(Gauss rhs) {
17         return {x * rhs.x - y * rhs.y, x * rhs.y + y * rhs.x};
18     }
19     Gauss operator/(Gauss rhs) {
20         auto [x, y] = operator*(~rhs);
21         return {div_round(x, rhs.norm()), div_round(y, rhs.norm())};
22     }
23     Gauss operator%(Gauss rhs) { return operator-(rhs*(operator/(rhs))); }
24 };

```

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

```

```

17         (a.first * b.second + a.second * b.first) % p);
18     };
19     pair<i64, i64> a = {x, 1}, res = {1, 0};
20     for (i64 r = (p + 1) >> 1; r; r >>= 1, a = mul(a, a)) {
21         if (r & 1) {
22             res = mul(res, a);
23         }
24     }
25     return res.first;
26 }

```

5.2.2 Logarithm

Find k such that $x^k \equiv y \pmod{n}$.

Constraints: $0 \leq x, y < n$.

```

1 i64 log(i64 x, i64 y, i64 n) {
2     if (y == 1 or n == 1) {
3         return 0;
4     }
5     if (not x) {
6         return y ? -1 : 1;
7     }
8     i64 res = 0, k = 1 % n;
9     for (i64 d; k != y and (d = gcd(x, n)) != 1; res += 1) {
10         if (y % d) {
11             return -1;
12         }
13         n /= d;
14         y /= d;
15         k = k * (x / d) % n;
16     }
17     if (k == y) {
18         return res;
19     }
20     unordered_map<i64, i64> mp;
21     i64 px = 1, m = sqrt(n) + 1;
22     for (int i = 0; i < m; i += 1, px = px * x % n) {
23         mp[y * px % n] = i;
24     }
25     i64 ppx = k * px % n;
26     for (int i = 1; i <= m; i += 1, ppx = ppx * px % n) {
27         if (mp.count(ppx)) {
28             return res + i * m - mp[ppx];
29         }
30     }
31     return -1;
32 }

```

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) {
14         return {};
15     }
16     i64 a = a0 / d * a1, b = (i128)(b1 - b0) / d * x % (a1 / d);
17     if (b < 0) {
18         b += a1 / d;
19     }
20     b = (i128)(a0 * b + b0) % a;
21     if (b < 0) {
22         b += a;
23     }
24     return {{a, b}};
25 }

```

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) {
4         return false;
5     }
6     if (n == 2) {
7         return true;
8     }
9     if (not(n % 2)) {
10         return false;
11     }
12     int r = countr_zero(u64(n - 1));
13     i64 d = (n - 1) >> r;
14     for (int pi : p) {
15         if (pi >= n) {
16             break;
17         }
18         i64 x = power(pi, d, n);
19         if (x == 1 or x == n - 1) {
20             continue;
21         }
22         for (int j = 1; j < r; j += 1) {
23             x = (i128)x * x % n;

```

```

24         if (x == n - 1) {
25             break;
26         }
27     }
28     if (x != n - 1) {
29         return false;
30     }
31 }
32 return true;
33 };

```

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) {
5         return {};
6     }
7     vector<i64> res;
8     function<void(i64)> rho = [&](i64 n) {
9         if (miller_rabin(n)) {
10             return res.push_back(n);
11         }
12         i64 d = n;
13         while (d == n) {
14             d = 1;
15             for (i64 k = 1, y = 0, x = 0, s = 1, c = uid(mt); d == 1;
16                 k <= 1, y = x, s = 1) {
17                 for (int i = 1; i <= k; i += 1) {
18                     x = ((i128)x * x + c) % n;
19                     s = (i128)s * abs(x - y) % n;
20                     if (not(i % 127) or i == k) {
21                         d = gcd(s, n);
22                         if (d != 1) {
23                             break;
24                         }
25                     }
26                 }
27             }
28             rho(d);
29             rho(n / d);
30         }
31     };
32     rho(n);
33     return res;
34 }

```

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) {
6         n = n / pi * (pi - 1);
7     }
8     return n;
9 }
10 i64 minimum_primitive_root(i64 n) {
11     i64 pn = phi(n);
12     auto pd = pollard_rho(pn);
13     ranges::sort(pd);
14     pd.erase(ranges::unique(pd).begin(), pd.end());
15     auto check = [&](i64 r) {
16         if (gcd(r, n) != 1) {
17             return false;
18         }
19         for (i64 pi : pd) {
20             if (power(r, pn / pi, n) == 1) {
21                 return false;
22             }
23         }
24         return true;
25     };
26     i64 r = 1;
27     while (not check(r)) {
28         r += 1;
29     }
30     return r;
31 }

```

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) {
10             break;
11         }
12         tie(n, m, a, b) = tuple(y / m, a, m, y % m);
13     }
14     return ans;
15 }

```

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) {
3         return b;
4     }
5     if (c % 2) {
6         if (b >= a) {
7             u64 t = (m - b + a - 1) / a;
8             u64 d = (t - 1) * p + q;
9             if (n <= d) {
10                 return b;
11             }
12             n -= d;
13             b += a * t - m;
14         }
15         b = a - 1 - b;
16     } else {
17         if (b < m - a) {
18             u64 t = (m - b - 1) / a;
19             u64 d = t * p;
20             if (n <= d) {
21                 return (n - 1) / p * a + b;
22             }
23             n -= d;
24             b += a * t;
25         }
26         b = m - 1 - b;
27     }
28     u64 d = m / a;
29     u64 res = min_of_mod(n, a, m % a, b, c += 1, (d - 1) * p + q, d * p + q);
30     return c % 2 ? m - 1 - res : a - 1 - res;
31 }
```

5.9 Stern Brocot Tree

```
1 struct Node {
2     int a, b;
3     vector<pair<int, char>> p;
4     Node(int a, int b)
5         : a(a), b(b) {
6         // gcd(a, b) == 1
7         while (a != 1 or b != 1) {
8             if (a > b) {
9                 int k = (a - 1) / b;
10                 p.emplace_back(k, 'R');
11                 a -= k * b;
12             } else {
13                 int k = (b - 1) / a;
14                 p.emplace_back(k, 'L');
```

```
15         b -= k * a;
16     }
17 }
18 }
19 Node(vector<pair<int, char>> p, int _a = 1, int _b = 1)
20     : p(p), a(_a), b(_b) {
21     for (auto [c, d] : p | views::reverse) {
22         if (d == 'R') {
23             a += c * b;
24         } else {
25             b += c * a;
26         }
27     }
28 }
29 };
```

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) {
23                         res ^= mem[i][j];
24                     }
25                 }
26             }
27         }
28         return res;
29     }
30 };
```

6 Numerical

6.1 Golden Search

```
1 template <int step>
2 f64 golden_search(function<f64(f64)> f, f64 l, f64 r) {
3     f64 ml = (numbers::phi - 1) * l + (2 - numbers::phi) * r;
4     f64 mr = l + r - ml;
5     f64 fml = f(ml), fmr = f(mr);
6     for (int i = 0; i < step; i += 1)
7         if (fml > fmr) {
8             l = ml;
9             ml = mr;
10            fml = fmr;
11            fmr = f(mr = (numbers::phi - 1) * r + (2 - numbers::phi) * l);
12        } else {
13            r = mr;
14            mr = ml;
15            fmr = fml;
16            fml = f(ml = (numbers::phi - 1) * l + (2 - numbers::phi) * r);
17        }
18    return midpoint(l, r);
19 }
```

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) {
11        return (sl + sr) + d / 15;
12    }
13    return adaptive_simpson(f, l, m, eps / 2) +
14           adaptive_simpson(f, m, r, eps / 2);
15 }
```

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



```

58     }
59     int k = -1;
60     for (int j = 0; j < m; j += 1) {
61         if (a[j][i] > eps) {
62             if (k == -1) {
63                 k = j;
64             } else {
65                 f64 d = b[j] * a[k][i] - b[k] * a[j][i];
66                 if (d < -eps or (d < eps and base[j] > base[k])) {
67                     k = j;
68                 }
69             }
70         }
71     }
72     if (k == -1) {
73         return false;
74     }
75     pivot(k, i);
76 }
77 return true;
78 }
79 vector<f64> x() const {
80     vector<f64> res(n);
81     for (int i = n; i < n + m; i += 1) {
82         if (base[i] < n) {
83             res[base[i]] = b[i - n];
84         }
85     }
86     return res;
87 }
88 };

```

6.4 Green's Theorem

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

6.5 Double Integral

$$\iint_D f(x, y) dx dy = \iint_D f(x(u, v), y(u, v)) \left| \frac{\partial x}{\partial u} \frac{\partial x}{\partial v} \right| du dv.$$

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]) {
9             swap(a[i], a[r[i]]);
10        }
11    }
12    for (int m = 1; m < n; m *= 2) {
13        complex<f64> wn(exp((inverse ? 1.i : -1.i) * numbers::pi / (f64)m));
14        for (int i = 0; i < n; i += m * 2) {
15            complex<f64> w = 1;
16            for (int j = 0; j < m; j += 1, w = w * wn) {
17                auto &x = a[i + j + m], &y = a[i + j], t = w * x;
18                tie(x, y) = pair(y - t, y + t);
19            }
20        }
21    }
22    if (inverse) {
23        for (auto& ai : a) {
24            ai /= n;
25        }
26    }
27 }

```

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]) {
9             swap(a[i], a[r[i]]);
10        }
11    }
12    for (int m = 1; m < n; m *= 2) {
13        i64 wn = power(inverse ? power(g, mod - 2) : g, (mod - 1) / m / 2);
14        for (int i = 0; i < n; i += m * 2) {
15            i64 w = 1;
16            for (int j = 0; j < m; j += 1, w = w * wn % mod) {
17                auto &x = a[i + j + m], &y = a[i + j], t = w * x % mod;
18                tie(x, y) = pair((y + mod - t) % mod, (y + t) % mod);
19            }
20        }
21    }
22    if (inverse) {
23        i64 inv = power(n, mod - 2);
24        for (auto& ai : a) {
25            ai = ai * inv % mod;
26        }
27    }

```

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>
2  T eps = 0;
3  template <>
4  f64 eps<f64> = 1e-9;
5  template <typename T>
6  int sign(T x) {
7      return x < -eps<T> ? -1 : x > eps<T>;
8  }
9  template <typename T>
10 struct P {
11     T x, y;
12     explicit P(T x = 0, T y = 0)
13         : x(x), y(y) {}
14     P operator*(T k) { return P(x * k, y * k); }
15     P operator+(P p) { return P(x + p.x, y + p.y); }
16     P operator-(P p) { return P(x - p.x, y - p.y); }
17     P operator-() { return P(-x, -y); }
18     T len2() { return x * x + y * y; }
19     T cross(P p) { return x * p.y - y * p.x; }
20     T dot(P p) { return x * p.x + y * p.y; }
21     bool operator==(P p) { return sign(x - p.x) == 0 and sign(y - p.y) == 0; }
22     int arg() { return y < 0 or (y == 0 and x > 0) ? -1 : x or y; }
23     P rotate90() { return P(-y, x); }
24 };
25 template <typename T>
26 bool argument(P<T> lhs, P<T> rhs) {
27     if (lhs.arg() != rhs.arg()) {
28         return lhs.arg() < rhs.arg();
29     }
30     return lhs.cross(rhs) > 0;
31 }
32 template <typename T>
33 struct L {
34     P<T> a, b;
35     explicit L(P<T> a = {}, P<T> b = {})
36         : a(a), b(b) {}
37     P<T> v() { return b - a; }
38     bool contains(P<T> p) {
39         return sign((p - a).cross(p - b)) == 0 and sign((p - a).dot(p - b)) <= 0;
40     }
41     int left(P<T> p) { return sign(v().cross(p - a)); }
42     optional<pair<T, T>> intersection(L l) {
43         auto y = v().cross(l.v());
44         if (sign(y) == 0) {
45             return {};
46         }
47         auto x = (l.a - a).cross(l.v());
48         return y < 0 ? pair(-x, -y) : pair(x, y);
49     }
50 };

```

```

51 template <typename T>
52 struct G {
53     vector<P<T>> g;
54     explicit G(int n)
55         : g(n) {}
56     explicit G(const vector<P<T>>& g)
57         : g(g) {}
58     optional<int> winding(P<T> p) {
59         int n = g.size(), res = 0;
60         for (int i = 0; i < n; i += 1) {
61             auto a = g[i], b = g[(i + 1) % n];
62             L l(a, b);
63             if (l.contains(p)) {
64                 return {};
65             }
66             if (sign(l.v().y) < 0 and l.left(p) >= 0) {
67                 continue;
68             }
69             if (sign(l.v().y) == 0) {
70                 continue;
71             }
72             if (sign(l.v().y) > 0 and l.left(p) <= 0) {
73                 continue;
74             }
75             if (sign(a.y - p.y) < 0 and sign(b.y - p.y) >= 0) {
76                 res += 1;
77             }
78             if (sign(a.y - p.y) >= 0 and sign(b.y - p.y) < 0) {
79                 res -= 1;
80             }
81         }
82         return res;
83     }
84     G convex() {
85         ranges::sort(g, {}, [&](P<T> p) { return pair(p.x, p.y); });
86         vector<P<T>> h;
87         for (auto p : g) {
88             while (ssize(h) >= 2 and
89                 sign((h.back() - h.end()[-2]).cross(p - h.back())) <= 0) {
90                 h.pop_back();
91             }
92             h.push_back(p);
93         }
94         int m = h.size();
95         for (auto p : g | views::reverse) {
96             while (ssize(h) > m and
97                 sign((h.back() - h.end()[-2]).cross(p - h.back())) <= 0) {
98                 h.pop_back();
99             }
100             h.push_back(p);
101         }
102         h.pop_back();
103         return G(h);

```

```

104     }
105     // Following function are valid only for convex.
106     T diameter2() {
107         int n = g.size();
108         T res = 0;
109         for (int i = 0, j = 1; i < n; i += 1) {
110             auto a = g[i], b = g[(i + 1) % n];
111             while (sign((b - a).cross(g[(j + 1) % n] - g[j])) > 0) {
112                 j = (j + 1) % n;
113             }
114             res = max(res, (a - g[j]).len2());
115             res = max(res, (a - g[j]).len2());
116         }
117         return res;
118     }
119     optional<bool> contains(P<T> p) {
120         if (g[0] == p) {
121             return {};
122         }
123         if (g.size() == 1) {
124             return false;
125         }
126         if (L(g[0], g[1]).contains(p)) {
127             return {};
128         }
129         if (L(g[0], g[1]).left(p) <= 0) {
130             return false;
131         }
132         if (L(g[0], g.back()).left(p) > 0) {
133             return false;
134         }
135         int i = *ranges::partition_point(views::iota(2, ssize(g)), [&](int i) {
136             return sign((p - g[0]).cross(g[i] - g[0])) <= 0;
137         });
138         int s = L(g[i - 1], g[i]).left(p);
139         if (s == 0) {
140             return {};
141         }
142         return s > 0;
143     }
144     int most(const function<P<T>(P<T>>& f) {
145         int n = g.size();
146         auto check = [&](int i) {
147             return sign(f(g[i]).cross(g[(i + 1) % n] - g[i])) >= 0;
148         };
149         P<T> f0 = f(g[0]);
150         bool check0 = check(0);
151         if (not check0 and check(n - 1)) {
152             return 0;
153         }
154         return *ranges::partition_point(views::iota(0, n), [&](int i) -> bool {
155             if (i == 0) {
156                 return true;

```

```

157     }
158     bool checki = check(i);
159     int t = sign(f0.cross(g[i] - g[0]));
160     if (i == 1 and checki == check0 and t == 0) {
161         return true;
162     }
163     return checki ^ (checki == check0 and t <= 0);
164 });
165 }
166 pair<int, int> tan(P<T> p) {
167     return {most([&](P<T> x) { return x - p; })),
168            most([&](P<T> x) { return p - x; })}};
169 }
170 pair<int, int> tan(L<T> l) {
171     return {most([&](P<T> _) { return l.v(); })),
172            most([&](P<T> _) { return -l.v(); })}};
173 }
174 };
175
176 template <typename T>
177 vector<L<T>> half(vector<L<T>> ls, T bound) {
178     // Ranges: bound ~ 6
179     auto check = [](L<T> a, L<T> b, L<T> c) {
180         auto [x, y] = b.intersection(c).value();
181         a = L(a.a * y, a.b * y);
182         return a.left(b.a * y + b.v() * x) < 0;
183     };
184     ls.emplace_back(P(-bound, (T)0), P(-bound, -(T)1));
185     ls.emplace_back(P((T)0, -bound), P((T)1, -bound));
186     ls.emplace_back(P(bound, (T)0), P(bound, (T)1));
187     ls.emplace_back(P((T)0, bound), P(-(T)1, bound));
188     ranges::sort(ls, [&](L<T> lhs, L<T> rhs) {

```

```

189         if (sign(lhs.v().cross(rhs.v())) == 0 and
190             sign(lhs.v().dot(rhs.v())) >= 0) {
191             return lhs.left(rhs.a) == -1;
192         }
193         return argument(lhs.v(), rhs.v());
194     });
195     deque<L<T>> q;
196     for (int i = 0; i < ssize(ls); i += 1) {
197         if (i and sign(ls[i - 1].v().cross(ls[i].v())) == 0 and
198             sign(ls[i - 1].v().dot(ls[i].v())) == 1) {
199             continue;
200         }
201         while (q.size() > 1 and check(ls[i], q.back(), q.end()[-2])) {
202             q.pop_back();
203         }
204         while (q.size() > 1 and check(ls[i], q[0], q[1])) {
205             q.pop_front();
206         }
207         if (not q.empty() and sign(q.back().v().cross(ls[i].v())) <= 0) {
208             return {};
209         }
210         q.push_back(ls[i]);
211     }
212     while (q.size() > 1 and check(q[0], q.back(), q.end()[-2])) {
213         q.pop_back();
214     }
215     while (q.size() > 1 and check(q.back(), q[0], q[1])) {
216         q.pop_front();
217     }
218     return vector<L<T>>(q.begin(), q.end());
219 }

```