

Team Reference Document

Heltion

March 8, 2024

Contents	
1 Contest	1
1.1 Makefile	1
1.2 debug.h	1
1.3 Template	1
1.4 Clang-foramt	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	10
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	17
5 Number Theory	18
5.1 Gaussian Integer	18
5.2 Modular Arithmetic	18
5.2.1 Sqrt	18
5.2.2 Logarithm	18
5.3 Chinese Remainder Theorem	19
5.4 Miller Rabin	19
5.5 Pollard Rho	19
5.6 Primitive Root	20
5.7 Sum of Floor	20
5.8 Minimum of Remainder	20
5.9 Stern Brocot Tree	21
5.10 Nim Product	21
6 Numerical	21
6.1 Golden Search	21
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	23
7.2.2 Arithmetic	23
7.2.3 Interpolation	23
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 \
3 -Wall -Wextra -Wconversion \
4 -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<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"
```

1.3 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 RBTree = 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 }
```

1.4 Clang-foramt

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(const vector<vector<int>>&
2 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);
```

```

29     return res;
30 }

```

2.1.2 Two-vertex-connected Components

```

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

```

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

```

```

28         if (deg[v] == 0) {
29             v = path[v];
30         }
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) {
47         dfs(i, -1);
48     }
49 }
50 vector<vector<int>> _res(n);
51 for (int i = 0; i < n; i += 1) {
52     _res[dsu.find(i)].push_back(i);
53 }
54 vector<vector<int>> res;
55 for (auto& res_i : _res) {
56     if (not res_i.empty()) {
57         res.emplace_back(move(res_i));
58     }
59 }
60 return res;
61 }

```

2.2 Euler Walks

```

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

```

```

15        return {};
16    }
17 }
18 vector<pair<int, bool>> walk;
19 vector<bool> visited(m);
20 vector<int> cur(n);
21 function<void(int)> dfs = [&](int u) {
22     for (int& i = cur[u]; i < ssize(g[u]);) {
23         auto [j, d] = g[u][i];
24         if (not visited[j]) {
25             visited[j] = true;
26             dfs(d ? edges[j].second : edges[j].first);
27             walk.emplace_back(j, d);
28         } else {
29             i += 1;
30         }
31     }
32 };
33 for (int i = 0; i < n; i += 1) {
34     dfs(i);
35     if (not walk.empty()) {
36         ranges::reverse(walk);
37         res.emplace_back(move(walk));
38     }
39 }
40 return res;
41 }
42 optional<vector<vector<int>>> directed_walks(int n, const vector<pair<int,
  int>>& edges) {
43     int m = ssize(edges);
44     vector<vector<int>> res;
45     if (not m) {
46         return res;
47     }
48     vector<int> d(n);
49     vector<vector<int>> g(n);
50     for (int i = 0; i < m; i += 1) {
51         auto [u, v] = edges[i];
52         g[u].push_back(i);
53         d[v] += 1;
54     }
55     for (int i = 0; i < n; i += 1) {
56         if (ssize(g[i]) != d[i]) {
57             return {};
58         }
59     }
60     vector<int> walk;
61     vector<int> cur(n);
62     vector<bool> visited(m);
63     function<void(int)> dfs = [&](int u) {
64         for (int& i = cur[u]; i < ssize(g[u]);) {
65             int j = g[u][i];
66             if (not visited[j]) {

```

```

67         visited[j] = true;
68         dfs(edges[j].second);
69         walk.push_back(j);
70     } else {
71         i += 1;
72     }
73 }
74 };
75 for (int i = 0; i < n; i += 1) {
76     dfs(i);
77     if (not walk.empty()) {
78         ranges::reverse(walk);
79         res.emplace_back(move(walk));
80     }
81 }
82 return res;
83 }

```

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) : 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) {
12            l->add(d);
13        }
14        if (r) {
15            r->add(d);
16        }
17        d = 0;
18    }
19 };
20 Node* merge(Node* u, Node* v) {
21     if (not u or not v) {
22         return u ? v :

```

```

23 }
24 if (u->e.w > v->e.w) {
25     swap(u, v);
26 }
27 u->push();
28 u->r = merge(u->r, v);
29 swap(u->l, u->r);
30 return u;
31 }
32 void pop(Node*& u) {
33     u->push();
34     u = merge(u->l, u->r);
35 }
36 pair<i64, vector<int>> directed_minimum_spanning_tree(int n, const vector<
    Edge>& edges, int s) {
37     i64 ans = 0;
38     vector<Node*> heap(n), edge(n);
39     RollbackDisjointSetUnion dsu(n), rbdsu(n);
40     vector<pair<Node*, int>> cycles;
41     for (auto e : edges) {
42         heap[e.v] = merge(heap[e.v], new Node(e));
43     }
44     for (int i = 0; i < n; i += 1) {
45         if (i == s) {
46             continue;
47         }
48         for (int u = i;;) {
49             if (not heap[u]) {
50                 return {};
51             }
52             ans += (edge[u] = heap[u])->e.w;
53             edge[u]->add(-edge[u]->e.w);
54             int v = rbdsu.find(edge[u]->e.u);
55             if (dsu.merge(u, v)) {
56                 break;
57             }
58             int t = rbdsu.time();
59             while (rbdsu.merge(u, v)) {
60                 heap[rbdsu.find(u)] = merge(heap[u], heap[v]);
61                 u = rbdsu.find(u);
62                 v = rbdsu.find(edge[v]->e.u);
63             }
64             cycles.emplace_back(edge[u], t);
65             while (heap[u] and rbdsu.find(heap[u]->e.u) == rbdsu.find(u)) {
66                 pop(heap[u]);
67             }
68         }
69     }
70     for (auto [p, t] : cycles | views::reverse) {
71         int u = rbdsu.find(p->e.v);
72         rbdsu.rollback(t);
73         int v = rbdsu.find(edge[u]->e.v);
74         edge[v] = exchange(edge[u], p);

```

```

75 }
76 vector<int> res(n, -1);
77 for (int i = 0; i < n; i += 1) {
78     res[i] = i == s ? i : edge[i]->e.u;
79 }
80 return {ans, res};
81 }

```

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

```

```

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

```

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 = ranges::min(views::iota(0, n), {}, [&](int r) { return w[r][c];
8             });
9         pi[c] = w[r][c];
10        if (rm[r] == -1) {
11            rm[r] = c;

```



```

11     cm[c] = r;
12 }
13 }
14 vector<int> cols(n);
15 iota(cols.begin(), cols.end(), 0);
16 for (int r = 0; r < n; r += 1) {
17     if (rm[r] != -1) {
18         continue;
19     }
20     vector<i64> d(n);
21     for (int c = 0; c < n; c += 1) {
22         d[c] = resid(r, c);
23     }
24     vector<int> pre(n, r);
25     int scan = 0, label = 0, last = 0, col = -1;
26     [&]() {
27         while (true) {
28             if (scan == label) {
29                 last = scan;
30                 i64 min = d[cols[scan]];
31                 for (int j = scan; j < n; j += 1) {
32                     int c = cols[j];
33                     if (d[c] <= min) {
34                         if (d[c] < min) {
35                             min = d[c];
36                             label = scan;
37                         }
38                         swap(cols[j], cols[label++]);
39                     }
40                 }
41                 for (int j = scan; j < label; j += 1) {
42                     if (int c = cols[j]; cm[c] == -1) {
43                         col = c;
44                         return;
45                     }
46                 }
47             }
48             int c1 = cols[scan++], r1 = cm[c1];
49             for (int j = label; j < n; j += 1) {
50                 int c2 = cols[j];
51                 i64 len = resid(r1, c2) - resid(r1, c1);
52                 if (d[c2] > d[c1] + len) {
53                     d[c2] = d[c1] + len;
54                     pre[c2] = r1;
55                     if (len == 0) {
56                         if (cm[c2] == -1) {
57                             col = c2;
58                             return;
59                         }
60                         swap(cols[j], cols[label++]);
61                     }
62                 }
63             }
64         }
65     }();
66     for (int i = 0; i < last; i += 1) {
67         int c = cols[i];
68         pi[c] += d[c] - d[col];
69     }
70     for (int t = col; t != -1;) {
71         col = t;
72         int r = pre[col];
73         cm[col] = r;
74         swap(rm[r], t);
75     }
76 }
77 i64 res = 0;
78 for (int i = 0; i < n; i += 1) {
79     res += w[i][rm[i]];
80 }
81 return {res, rm};
82 }

```

```

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

```

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

```

```

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

```

```

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

```

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]) {
29                 continue;
30             }
31             for (int i : g[u]) {
32                 auto [_, v, f, c] = edges[i];
33                 if (f and d[v] > d[u] + h[u] - h[v] + c) {
34                     p[v] = i;
35                     q.emplace(d[v] = d[u] + h[u] - h[v] + c, v);
36                 }
37             }
38         }
39         return ~p[t];
40     };
41     i64 f = 0, c = 0;
42     while (dijkstra()) {
43         for (int i = 0; i < n; i += 1) {
44             h[i] += d[i];
45         }
46         vector<int> path;
47         for (int u = t; u != s; u = edges[p[u]].u) {
48             path.push_back(p[u]);
49         }
50         i64 mf = edges[ranges::min(path, {}, [&](int i) { return edges[i].f; })
51             ].f;
52         f += mf;
53         c += mf * h[t];
54         for (int i : path) {
55             edges[i].f -= mf;
56             edges[i ^ 1].f += mf;
57         }
58     }
59     return {f, c};
60 };

```

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

```

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) : 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) {
22         return {};
23     }
24     if (p->v < v) {
25         auto [l, r] = split_by_v(p->r, v);
26         return {p->update(p->l, l), r};
27     }
28     auto [l, r] = split_by_v(p->l, v);
29     return {l, p->update(r, p->r)};
30 }
31 pair<Node*, Node*> split_by_size(Node* p, int size) {
32     if (not p) {
33         return {};
34     }
35     int l_size = p->l ? p->l->size : 0;
36     if (l_size < size) {
37         auto [l, r] = split_by_size(p->r, size - l_size - 1);
38         return {p->update(p->l, l), r};
39     }
40     auto [l, r] = split_by_size(p->l, size);
41     return {l, p->update(r, p->r)};
42 }
43 Node* merge(Node* l, Node* r) {
44     if (not l or not r) {
45         return l ? r;
46     }
47     if (l->priority < r->priority) {
48         return r->update(merge(l, r->l), r->r);
49     }
50     return l->update(l->l, merge(l->r, r));
51 }

```

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) : 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, i64
            tr, Segment s) {

```

```

15         if (p == nullptr) {
16             p = new Node();
17         }
18         i64 tm = midpoint(tl, tr);
19         if (tl >= l and tr <= r) {
20             if (not p->s) {
21                 p->s = s;
22                 return;
23             }
24             auto t = p->s.value();
25             if (t.get(tl) >= s.get(tl)) {
26                 if (t.get(tr) >= s.get(tr)) {
27                     return;
28                 }
29                 if (t.get(tm) >= s.get(tm)) {
30                     return rec(p->r, tm + 1, tr, s);
31                 }
32                 p->s = s;
33                 return rec(p->l, tl, tm, t);
34             }
35             if (t.get(tr) <= s.get(tr)) {
36                 p->s = s;
37                 return;
38             }
39             if (t.get(tm) <= s.get(tm)) {
40                 p->s = s;
41                 return rec(p->r, tm + 1, tr, t);
42             }
43             return rec(p->l, tl, tm, s);
44         }
45         if (l <= tm) {
46             rec(p->l, tl, tm, s);
47         }
48         if (r > tm) {
49             rec(p->r, tm + 1, tr, s);
50         }
51     };
52     rec(root, tl, tr, {k, b});
53 }
54 optional<i64> get(i64 x) {
55     optional<i64> res = {};
56     function<void(Node*, i64, i64)> rec = [&](Node* p, i64 tl, i64 tr) {
57         if (p == nullptr) {
58             return;
59         }
60         i64 tm = midpoint(tl, tr);
61         if (p->s) {
62             i64 y = p->s.value().get(x);
63             if (not res or res.value() < y) {
64                 res = y;
65             }
66         }
67         if (x <= tm) {

```

```

68     rec(p->l, tl, tm);
69 } else {
70     rec(p->r, tm + 1, tr);
71 }
72 };
73 rec(root, tl, tr);
74 return res;
75 }
76 };

```

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()) {
20            smv += x;
21        }
22        if (tmv != def()) {
23            tmv += x;
24        }
25    }
26 };
27 struct Node {
28     Mv mn, mx;
29     i64 sum, tsum;
30     Node *ls, *rs;
31     Node(i64 x = 0) : sum(x), tsum(0), mn(x, true), mx(x, false) { ls = rs = nullptr; }
32     void up() {
33         sum = ls->sum + rs->sum;
34         mx.up(ls->mx, rs->mx);
35         mn.up(ls->mn, rs->mn);
36     }
37     void down(int tl, int tr) {
38         if (tsum) {
39             int tm = midpoint(tl, tr);

```

```

40         ls->add(tl, tm, tsum);
41         rs->add(tm, tr, tsum);
42         tsum = 0;
43     }
44     if (mn.tmv != mn.def()) {
45         ls->ch(mn.tmv, true);
46         rs->ch(mn.tmv, true);
47         mn.tmv = mn.def();
48     }
49     if (mx.tmv != mx.def()) {
50         ls->ch(mx.tmv, false);
51         rs->ch(mx.tmv, false);
52         mx.tmv = mx.def();
53     }
54 }
55 bool cmp(i64 x, i64 y, bool less) { return less ? x < y : x > y; }
56 void add(int tl, int tr, i64 x) {
57     sum += (tr - tl) * x;
58     tsum += x;
59     mx.add(x);
60     mn.add(x);
61 }
62 void ch(i64 x, bool less) {
63     auto &lhs = less ? mn : mx, &rhs = less ? mx : mn;
64     if (not cmp(x, rhs.mv, less)) {
65         return;
66     }
67     sum += (x - rhs.mv) * rhs.cmv;
68     if (lhs.smv == rhs.mv) {
69         lhs.smv = x;
70     }
71     if (lhs.mv == rhs.mv) {
72         lhs.mv = x;
73     }
74     if (cmp(x, rhs.tmv, less)) {
75         rhs.tmv = x;
76     }
77     rhs.mv = lhs.tmv = x;
78 }
79 void add(int tl, int tr, int l, int r, i64 x) {
80     if (tl >= l and tr <= r) {
81         return add(tl, tr, x);
82     }
83     down(tl, tr);
84     int tm = midpoint(tl, tr);
85     if (l < tm) {
86         ls->add(tl, tm, l, r, x);
87     }
88     if (r > tm) {
89         rs->add(tm, tr, l, r, x);
90     }
91     up();
92 }

```

```

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

```

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    dfs(root);
18    a.resize(n);
19    for (int i = 1; i < n; i += 1) {
20        a[i] = time[par[p[i]]];
21    }
22    st = SparseTable(a);
23 }
24 int query(int u, int v) {
25     if (u == v) {
26         return u;
27     }
28     if (time[u] > time[v]) {
29         swap(u, v);
30     }
31     return p[st.query(time[u] + 1, time[v] + 1)];
32 }
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()) : t(t), st(t), reversed(false), par(nullptr) { ch.fill(
8         nullptr); }
9     int get_s() {
10         if (par == nullptr) {
11             return -1;
12         }
13         if (par->ch[0] == this) {
14             return 0;
15         }
16         if (par->ch[1] == this) {
17             return 1;
18         }
19         return -1;
20     }
21     void push_up() { st = OP()(ch[0] ? ch[0]->st : E(), OP()(t, ch[1] ? ch
22         [1]->st : E())); }
23     void reverse() {
24         reversed ^= 1;
25         st = REV()(st);
26     }
27     void push_down() {
28         if (reversed) {

```



```

27     swap(ch[0], ch[1]);
28     if (ch[0]) {
29         ch[0]->reverse();
30     }
31     if (ch[1]) {
32         ch[1]->reverse();
33     }
34     reversed = false;
35 }
36 }
37 void attach(int s, Node* u) {
38     if ((ch[s] = u)) {
39         u->par = this;
40     }
41     push_up();
42 }
43 void rotate() {
44     auto p = par;
45     auto pp = p->par;
46     int s = get_s();
47     int ps = p->get_s();
48     p->attach(s, ch[s ^ 1]);
49     attach(s ^ 1, p);
50     if (~ps) {
51         pp->attach(ps, this);
52     }
53     par = pp;
54 }
55 void splay() {
56     push_down();
57     while (~get_s() and ~par->get_s()) {
58         par->par->push_down();
59         par->push_down();
60         push_down();
61         (get_s() == par->get_s() ? par : this)->rotate();
62         rotate();
63     }
64     if (~get_s()) {
65         par->push_down();
66         push_down();
67         rotate();
68     }
69 }
70 void access() {
71     splay();
72     attach(1, nullptr);
73     while (par != nullptr) {
74         auto p = par;
75         p->splay();
76         p->attach(1, this);
77         rotate();
78     }
79 }

```

```

80 void make_root() {
81     access();
82     reverse();
83     push_down();
84 }
85 void link(Node* u) {
86     u->make_root();
87     access();
88     attach(1, u);
89 }
90 void cut(Node* u) {
91     u->make_root();
92     access();
93     if (ch[0] == u) {
94         ch[0] = u->par = nullptr;
95         push_up();
96     }
97 }
98 void set(T t) {
99     access();
100    this->t = t;
101    push_up();
102 }
103 T query(Node* u) {
104     u->make_root();
105     access();
106     return st;
107 }
108 };

```

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]++;
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]++;
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 previous[p[i] + m] == previous[p[i - 1] + m] ? rank[p[i - 1]] : k++;
30        }
31    }
32    vector<int> lcp(n);
33    k = 0;
34    for (int i = 0; i < n; i++) {
35        if (rank[i]) {
36            k = max(k - 1, 0);
37            int j = p[rank[i] - 1];
38            while (i + k < n and j + k < n and s[i + k] == s[j + k]) {
39                k++;
40            }
41            lcp[rank[i]] = k;
42        }
43    }
44    return {p, lcp};
45 }
```

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]) {
25                 q.push(at(0).next[i]);
26             }
27         }
28         while (not q.empty()) {
29             int u = q.front();
30             q.pop();
31             for (int i = 0; i < sigma; i += 1) {
32                 if (at(u).next[i]) {
33                     at(at(u).next[i]).link = at(at(u).link).next[i];
34                     q.push(at(u).next[i]);
35                 } else {
36                     at(u).next[i] = at(at(u).link).next[i];
37                 }
38             }
39         }
40     }
41 };
```

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) {
13                 return q;
14             }
15             int clone = size();
16             push_back(at(q));
17             back().len = at(p).len + 1;
18             while (~p and at(p).next[c] == q) {
19                 at(p).next[c] = clone;
20                 p = at(p).link;
21             }
22             at(q).link = clone;
23             return clone;
24         }
25         int cur = size();
26         emplace_back();
27         back().len = at(p).len + 1;
28         while (~p and at(p).next[c] == -1) {
29             at(p).next[c] = cur;
30             p = at(p).link;
31         }
32         if (~p) {
33             int q = at(p).next[c];
34             if (at(p).len + 1 == at(q).len) {
35                 back().link = q;
36             } else {
37                 int clone = size();
38                 push_back(at(q));
39                 back().len = at(p).len + 1;
40                 while (~p and at(p).next[c] == q) {
41                     at(p).next[c] = clone;
42                     p = at(p).link;
43                 }
44                 at(q).link = at(cur).link = clone;
45             }
46         } else {
47             back().link = 0;
48         }
49         return cur;
50     }
51 };
```

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])
19             u = at(u).link;
20         return u;
21     }
22     void extend(int i) {
23         int cur = get_link(last, i);
24         if (not at(cur).next[s[i]]) {
25             int now = size();
26             emplace_back(at(cur).len + 2);
27             back().link = at(get_link(at(cur).link, i)).next[s[i]];
28             back().sum = at(back().link).sum + 1;
29             at(cur).next[s[i]] = now;
30         }
31         last = at(cur).next[s[i]];
32     }
33 };

```

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) { return {x * rhs.x - y * rhs.y, x * rhs.y + y *
17          rhs.x}; }
18      Gauss operator/(Gauss rhs) {
19          auto [x, y] = operator*(~rhs);
20          return {div_round(x, rhs.norm()), div_round(y, rhs.norm())};
21      }
22      Gauss operator%(Gauss rhs) { return operator-(rhs*(operator/(rhs))); }
23  };

```

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

```

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; k <= 1, y =
           x, s = 1) {
16           for (int i = 1; i <= k; i += 1) {
17               x = ((i128)x * x + c) % n;
18               s = (i128)s * abs(x - y) % n;
19               if (not(i % 127) or i == k) {
20                   d = gcd(s, n);
21                   if (d != 1) {
22                       break;
23                   }
24               }
25           }
26       }
27   }
28   rho(d);
29   rho(n / d);
30 };
31 rho(n);
32 return res;
33 }

```

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       }
16       b = a - 1 - b;
17   } else {
18       if (b < m - a) {
19           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) : 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) : p(p), a(_a), b(_b) {
19        for (auto [c, d] : p | views::reverse) {
20            if (d == 'R') {
21                a += c * b;
22            } else {
23                b += c * a;
24            }
25        }
26    }
27 };

```

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] ^ mem[(i ^ x) | (x - 1)][(j ^ x) | (i & (x - 1))];
12                }
13            }
14        }
15    }
16    u64 nim_product(u64 x, u64 y) {
17        u64 res = 0;
18        for (int i = 0; i < 64 and x >> i; i += 1) {
19            if ((x >> i) % 2) {
20                for (int j = 0; j < 64 and y >> j; j += 1) {
21                    if ((y >> j) % 2) {
22                        res ^= mem[i][j];
23                    }
24                }
25            }
26        }
27        return res;
28    }
29 };

```

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) + adaptive_simpson(f, m, r, eps /
14    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) : n(n), m(m), a(m, vector<f64>(n)), b(m), c(n), base(
8     n + m), z(0) { iota(base.begin(), base.end(), 0); }
9     void pivot(int out, int in) {
10        swap(base[out + n], base[in]);
11        f64 f = 1 / a[out][in];
12        for (f64& aij : a[out]) {
13            aij *= f;
14        }
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) {
24                        ai[j] += a[out][j] * f;
25                    }
26                    ai[in] = a[out][in] * f;
27                    bi += b[out] * f;
28                }
29            }
30        }
31    }
```

```
30 }
31 bool feasible() {
32     while (true) {
33         int i = ranges::min_element(b) - b.begin();
34         if (b[i] > -eps) {
35             break;
36         }
37         int k = -1;
38         for (int j = 0; j < n; j += 1) {
39             if (a[i][j] < -eps and (k == -1 or base[j] > base[k])) {
40                 k = j;
41             }
42         }
43         if (k == -1) {
44             return false;
45         }
46         pivot(i, k);
47     }
48     return true;
49 }
50 bool bounded() {
51     while (true) {
52         int i = ranges::max_element(c) - c.begin();
53         if (c[i] < eps) {
54             break;
55         }
56         int k = -1;
57         for (int j = 0; j < m; j += 1) {
58             if (a[j][i] > eps) {
59                 if (k == -1) {
60                     k = j;
61                 } else {
62                     f64 d = b[j] * a[k][i] - b[k] * a[j][i];
63                     if (d < -eps or (d < eps and base[j] > base[k])) {
64                         k = j;
65                     }
66                 }
67             }
68         }
69         if (k == -1) {
70             return false;
71         }
72         pivot(k, i);
73     }
74     return true;
75 }
76 vector<f64> x() const {
77     vector<f64> res(n);
78     for (int i = n; i < n + m; i += 1) {
79         if (base[i] < n) {
80             res[base[i]] = b[i - n];
81         }
82     }
83 }
```



```

83     return res;
84 }
85 };

```

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    }
28 }

```

7.2.1 Newton's Method

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

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

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

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

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) : x(x), y(y) {}
13     P operator*(T k) { return P(x * k, y * k); }
14     P operator+(P p) { return P(x + p.x, y + p.y); }
15     P operator-(P p) { return P(x - p.x, y - p.y); }
16     P operator-() { return P(-x, -y); }
17     T len2() { return x * x + y * y; }
18     T cross(P p) { return x * p.y - y * p.x; }
19     T dot(P p) { return x * p.x + y * p.y; }
20     bool operator==(P p) { return sign(x - p.x) == 0 and sign(y - p.y) == 0; }
21     int arg() { return y < 0 or (y == 0 and x > 0) ? -1 : x or y; }
22     P rotate90() { return P(-y, x); }
23 };
24 template <typename T>

```

```

25 bool argument(P<T> lhs, P<T> rhs) {
26     if (lhs.arg() != rhs.arg()) {
27         return lhs.arg() < rhs.arg();
28     }
29     return lhs.cross(rhs) > 0;
30 }
31 template <typename T>
32 struct L {
33     P<T> a, b;
34     explicit L(P<T> a = {}, P<T> b = {}) : a(a), b(b) {}
35     P<T> v() { return b - a; }
36     bool contains(P<T> p) { return sign((p - a).cross(p - b)) == 0 and sign((p
    - a).dot(p - b)) <= 0; }
37     int left(P<T> p) { return sign(v().cross(p - a)); }
38     optional<pair<T, T>> intersection(L l) {
39         auto y = v().cross(l.v());
40         if (sign(y) == 0) {
41             return {};
42         }
43         auto x = (l.a - a).cross(l.v());
44         return y < 0 ? pair(-x, -y) : pair(x, y);
45     }
46 };
47 template <typename T>
48 struct G {
49     vector<P<T>> g;
50     explicit G(int n) : g(n) {}
51     explicit G(const vector<P<T>>& g) : g(g) {}
52     optional<int> winding(P<T> p) {
53         int n = g.size(), res = 0;
54         for (int i = 0; i < n; i += 1) {
55             auto a = g[i], b = g[(i + 1) % n];
56             L l(a, b);
57             if (l.contains(p)) {
58                 return {};
59             }
60             if (sign(l.v().y) < 0 and l.left(p) >= 0) {
61                 continue;
62             }
63             if (sign(l.v().y) == 0) {
64                 continue;
65             }
66             if (sign(l.v().y) > 0 and l.left(p) <= 0) {
67                 continue;
68             }
69             if (sign(a.y - p.y) < 0 and sign(b.y - p.y) >= 0) {
70                 res += 1;
71             }
72             if (sign(a.y - p.y) >= 0 and sign(b.y - p.y) < 0) {
73                 res -= 1;
74             }
75         }
76         return res;

```

```

77 }
78 G convex() {
79     ranges::sort(g, {}, [&](P<T> p) { return pair(p.x, p.y); });
80     vector<P<T>> h;
81     for (auto p : g) {
82         while (ssize(h) >= 2 and sign((h.back() - h.end()[-2]).cross(p - h.back
83             ())) <= 0) {
84             h.pop_back();
85         }
86         h.push_back(p);
87     }
88     int m = h.size();
89     for (auto p : g | views::reverse) {
90         while (ssize(h) > m and sign((h.back() - h.end()[-2]).cross(p - h.back
91             ())) <= 0) {
92             h.pop_back();
93         }
94         h.push_back(p);
95     }
96     h.pop_back();
97     return G(h);
98 }
99 // Following function are valid only for convex.
100 T diameter2() {
101     int n = g.size();
102     T res = 0;
103     for (int i = 0, j = 1; i < n; i += 1) {
104         auto a = g[i], b = g[(i + 1) % n];
105         while (sign((b - a).cross(g[(j + 1) % n] - g[j]))) > 0) {
106             j = (j + 1) % n;
107         }
108         res = max(res, (a - g[j]).len2());
109         res = max(res, (a - g[j]).len2());
110     }
111     return res;
112 }
113 optional<bool> contains(P<T> p) {
114     if (g[0] == p) {
115         return true;
116     }
117     if (g.size() == 1) {
118         return false;
119     }
120     if (L(g[0], g[1]).contains(p)) {
121         return true;
122     }
123     if (L(g[0], g.back()).left(p) > 0) {
124         return false;
125     }
126     int i = *ranges::partition_point(views::iota(2, ssize(g)), [&](int i) {
127
128         return sign((p - g[0]).cross(g[i] - g[0])) <= 0; });
129     int s = L(g[i - 1], g[i]).left(p);
130     if (s == 0) {
131         return true;
132     }
133     return s > 0;
134 }
135 int most(const function<P<T>(P<T>)>& f) {
136     int n = g.size();
137     auto check = [&](int i) { return sign(f(g[i]).cross(g[(i + 1) % n] - g[i
138         ])) >= 0; };
139     P<T> f0 = f(g[0]);
140     bool check0 = check(0);
141     if (not check0 and check(n - 1)) {
142         return 0;
143     }
144     return *ranges::partition_point(views::iota(0, n), [&](int i) -> bool {
145         if (i == 0) {
146             return true;
147         }
148         bool checki = check(i);
149         int t = sign(f0.cross(g[i] - g[0]));
150         if (i == 1 and checki == check0 and t == 0) {
151             return true;
152         }
153         return checki ^ (checki == check0 and t <= 0);
154     });
155 }
156 pair<int, int> tan(P<T> p) {
157     return {most([&](P<T> x) { return x - p; }), most([&](P<T> x) { return p
158         - x; })};
159 }
160 pair<int, int> tan(L<T> l) {
161     return {most([&](P<T> _) { return l.v(); }), most([&](P<T> _) { return -l
162         .v(); })};
163 }
164 }
165 template <typename T>
166 vector<L<T>> half(vector<L<T>> ls, T bound) {
167     // Ranges: bound ~ 6
168     auto check = [&](L<T> a, L<T> b, L<T> c) {
169         auto [x, y] = b.intersection(c).value();
170         a = L(a.a * y, a.b * y);
171         return a.left(b.a * y + b.v() * x) < 0;
172     };
173     ls.emplace_back(P(-bound, (T)0), P(-bound, -(T)1));
174     ls.emplace_back(P((T)0, -bound), P((T)1, -bound));
175     ls.emplace_back(P(bound, (T)0), P(bound, (T)1));
176     ls.emplace_back(P((T)0, bound), P(-(T)1, bound));
177     ranges::sort(ls, [&](L<T> lhs, L<T> rhs) {
178         if (sign(lhs.v().cross(rhs.v())) == 0 and sign(lhs.v().dot(rhs.v())) >=
179             0) {
180

```

```

176     return lhs.left(rhs.a) == -1;
177 }
178 return argument(lhs.v(), rhs.v());
179 });
180 deque<L<T>> q;
181 for (int i = 0; i < ssize(ls); i += 1) {
182     if (i and sign(ls[i - 1].v().cross(ls[i].v())) == 0 and sign(ls[i - 1].v
        ().dot(ls[i].v())) == 1) {
183         continue;
184     }
185     while (q.size() > 1 and check(ls[i], q.back(), q.end()[-2])) {
186         q.pop_back();
187     }
188     while (q.size() > 1 and check(ls[i], q[0], q[1])) {
189         q.pop_front();

```

```

190     }
191     if (not q.empty() and sign(q.back().v().cross(ls[i].v())) <= 0) {
192         return {};
193     }
194     q.push_back(ls[i]);
195 }
196 while (q.size() > 1 and check(q[0], q.back(), q.end()[-2])) {
197     q.pop_back();
198 }
199 while (q.size() > 1 and check(q.back(), q[0], q[1])) {
200     q.pop_front();
201 }
202 return vector<L<T>>(q.begin(), q.end());
203 }

```