

Team Reference Document

Heltion

March 8, 2024

Contents

1	Contest	1	5	Number Theory	18
1.1	Makefile	1	5.1	Gaussian Integer	18
1.2	debug.h	1	5.2	Modular Arithmetic	18
1.3	Template	1	5.2.1	Sqrt	18
2	Graph	1	5.2.2	Logarithm	19
2.1	Connected Components	1	5.3	Chinese Remainder Theorem	19
2.1.1	Strongly Connected Components . .	1	5.4	Miller Rabin	19
2.1.2	Two-vertex-connected Components .	2	5.5	Pollard Rho	20
2.1.3	Two-edge-connected Components . .	2	5.6	Primitive Root	20
2.1.4	Three-edge-connected Components .	2	5.7	Sum of Floor	20
2.2	Euler Walks	3	5.8	Minimum of Remainder	21
2.3	Dominator Tree	4	5.9	Stern Brocot Tree	21
2.4	Directed Minimum Spanning Tree	4	5.10	Nim Product	21
2.5	K Shortest Paths	5	6	Numerical	22
2.6	Global Minimum Cut	6	6.1	Golden Search	22
2.7	Minimum Perfect Matching on Bipartite Graph	6	6.2	Adaptive Simpson	22
2.8	Matching on General Graph	7	6.3	Simplex	22
2.9	Maximum Flow	8	6.4	Green's Theorem	23
2.10	Minimum Cost Maximum Flow	9	6.5	Double Integral	23
3	Data Structure	10	7	Convolution	23
3.1	Disjoint Set Union	10	7.1	Fast Fourier Transform on \mathbb{C}	23
3.2	Sparse Table	11	7.2	Formal Power Series on \mathbb{F}_p	23
3.3	Treap	11	7.2.1	Newton's Method	24
3.4	Lines Maximum	11	7.2.2	Arithmetic	24
3.5	Segments Maximum	12	7.2.3	Interpolation	24
3.6	Segment Beats	13	7.2.4	Primes with root 3	24
3.7	Tree	14	7.3	Circular Transform	24
3.7.1	Least Common Ancestor	14	7.4	Truncated Transform	24
3.7.2	Link Cut Tree	14	8	Geometry	24
4	String	15	8.1	Pick's Theorem	24
4.1	Z	15	8.2	2D Geometry	24
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			

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<ostringstream>(ostringstream
10        () << x).str();
11 }
12 string to_debug(ranges::range auto x, string s =
13     "")
14     requires(not is_same_v<decltype(x), string>)
15 {
16     for (auto xi : x) { s += ", " + to_debug(xi); }
17     return "[" + s.substr(s.empty() ? 0 : 2) + "]";
18 }
19 template <class T, size_t size>
20 string to_debug(T x, string s)
21     requires(not ranges::range<T>)
22 {
23     [&<size_t... I>(index_sequence<I...>) {
24         ((s += ", " + to_debug(get<I>(x))), ...);
25     }(make_index_sequence<size>());
26     return "(" + s.substr(s.empty() ? 0 : 2) + ")";
27 }
28 #define debug(...) \
    cerr << __FILE__ ":" << __LINE__ \
    << ": " << (" #__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     >;
16 using i64 = int64_t;
17 int main() {
18     cin.tie(nullptr)->sync_with_stdio(false);
19     cout << fixed << setprecision(20);
20 }
```

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
2     (
3     const vector<vector<int>>& g) {
4     int n = g.size();
5     vector<bool> done(n);
6     vector<int> pos(n, -1), stack;
7     vector<vector<int>> res;
8     function<int(int)> dfs = [&](int u) {
9         int low = pos[u] = stack.size();
10        stack.push_back(u);
11        for (int v : g[u]) {
12            if (not done[v]) {
13                low = min(low, ~pos[v] ? pos[v] : dfs(v));
14            }
15        }
16        if (low == pos[u]) {
17            res.emplace_back(stack.begin() + low, stack
18                .end());
19            for (int v : res.back()) {
20                done[v] = true;
21            }
22            stack.resize(low);
23        }
24    };
25    for (int u = 0; u < n; u++) {
26        if (not done[u]) {
27            dfs(u);
28        }
29    }
30    return res;
31 }
```

```
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>>
2     two_vertex_connected_components(
3     const vector<vector<int>>& g) {
4     int n = g.size();
5     vector<int> pos(n, -1), stack;
6     vector<vector<int>> res;
7     function<int(int, int)> dfs = [&](int u, int p)
8     {
9         int low = pos[u] = stack.size(), son = 0;
10        stack.push_back(u);
11        for (int v : g[u]) {
12            if (v != p) {
13                if (~pos[v]) {
14                    low = min(low, pos[v]);
15                } else {
16                    int end = stack.size(), lowv = dfs(v, u);
17                    low = min(low, lowv);
18                    if (lowv >= pos[u] and (~p or son++)) {
19                        res.emplace_back(stack.begin() + end,
20                            stack.end());
21                        res.back().push_back(u);
22                        stack.resize(end);
23                    }
24                }
25            }
26        }
27        return low;
28    };
29    for (int i = 0; i < n; i += 1) {
30        if (pos[i] == -1) {
31            dfs(i, -1);
32            res.emplace_back(move(stack));
33        }
34    }
35    return res;
36 }
```

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

2.1.4 Three-edge-connected Components

```
1 vector<vector<int>>
2   three_edge_connected_components(
3   const vector<vector<int>>& g) {
4   int n = g.size(), dft = -1;
5   vector<int> pre(n, -1), post(n), path(n, -1),
6   low(n), deg(n);
7   DisjointSetUnion dsu(n);
8   function<void(int, int)> dfs = [&](int u, int p
9   ) {
10       int pc = 0;
11       low[u] = pre[u] = dft += 1;
12       for (int v : g[u]) {
13          if (v != u and (v != p or pc++)) {
14              if (pre[v] != -1) {
15                  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
19              pre[v] <= post[p];) {
20              dsu.merge(u, p);
21              deg[u] += deg[p];
22              p = path[p];
23          }
24      }
25      } else {
26          dfs(v, u);
27          if (path[v] == -1 and deg[v] <= 1) {
28              low[u] = min(low[u], low[v]);
29              deg[u] += deg[v];
30          } else {
31              if (deg[v] == 0) {
32                  v = path[v];
33              }
34              if (low[u] > low[v]) {
35                  low[u] = min(low[u], low[v]);
36                  swap(v, path[u]);
37              }
38              for (; v != -1; v = path[v]) {
39                  dsu.merge(u, v);
40                  deg[u] += deg[v];
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>>>>
2   undirected_walks(
3   int n,
4   const vector<pair<int, int>>& edges) {
5   int m = ssize(edges);
6   vector<vector<pair<int, bool>>> res;
7   if (not m) {
8       return res;
9   }
10  vector<vector<pair<int, bool>>> g(n);
11  for (int i = 0; i < m; i += 1) {
12      auto [u, v] = edges[i];
13      g[u].emplace_back(i, true);
14      g[v].emplace_back(i, false);
15  }
16  for (int i = 0; i < n; i += 1) {
17      if (g[i].size() % 2) {
18          return {};
19      }
20  }
21  vector<pair<int, bool>> walk;
22  vector<bool> visited(m);
23  vector<int> cur(n);
24  function<void(int)> dfs = [&](int u) {
25      for (int& i = cur[u]; i < ssize(g[u]);) {
26          auto [j, d] = g[u][i];
27          if (not visited[j]) {
28              visited[j] = true;
29              dfs(d ? edges[j].second : edges[j].first);
30              ;
31              walk.emplace_back(j, d);
32          } else {
33              i += 1;
34          }
35      }
36      return res;
37  };
38  for (int i = 0; i < n; i += 1) {
39      dfs(i);
40      if (not walk.empty()) {
41          ranges::reverse(walk);
42          res.emplace_back(move(walk));
43      }
44  }
45  return res;
46  }
47  optional<vector<vector<int>>> directed_walks(
48  int n,
49  const vector<pair<int, int>>& edges) {
50  int m = ssize(edges);
51  vector<vector<int>> res;
52  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>>&
2     g, int s) {
3     int n = g.size();
4     vector<int> pos(n, -1), p, label(n), dom(n),
5         sdom(n), dsu(n), par(n);
6     vector<vector<int>> rg(n), bucket(n);
7     function<void(int)> dfs = [&](int u) {
8         int t = p.size();
9         p.push_back(u);
10        label[t] = sdom[t] = dsu[t] = pos[u] = t;
11        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 ) {
19     if (u == dsu[u]) {
20         return x ? -1 : u;
21     }
22     int v = find(dsu[u], x + 1);
23     if (v < 0) {
24         return u;
25     }
26     if (sdom[label[dsu[u]]] < sdom[label[u]]) {
27         label[u] = label[dsu[u]];
28     }
29     dsu[u] = v;
30     return x ? v : label[u];
31 };
32 dfs(s);
33 iota(dom.begin(), dom.end(), 0);
34 for (int i = ssize(p) - 1; i >= 0; i -= 1) {
35     for (int j : rg[i]) {
36         sdom[i] = min(sdom[i], sdom[find(j, 0)]);
37     }
38     if (i) {
39         bucket[sdom[i]].push_back(i);
40     }
41     for (int k : bucket[i]) {
42         int j = find(k, 0);
43         dom[k] = sdom[j] == sdom[k] ? sdom[j] : j;
44     }
45     if (i > 1) {
46         dsu[i] = par[i];
47     }
48 }
49 for (int i = 1; i < ssize(p); i += 1) {
50     if (dom[i] != sdom[i]) {
51         dom[i] = dom[dom[i]];
52     }
53 }
54 vector<int> res(n, -1);
55 res[s] = s;
56 for (int i = 1; i < ssize(p); i += 1) {
57     res[p[i]] = p[dom[i]];
58 }
59 return res;
60 }

```

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
39     vector<Edge>& edges, int s) {
40     i64 ans = 0;
41     vector<Node*> heap(n), edge(n);
42     RollbackDisjointSetUnion dsu(n), rbdsu(n);
43     vector<pair<Node*, int>> cycles;
44     for (auto e : edges) {
45         heap[e.v] = merge(heap[e.v], new Node(e));
46     }
47     for (int i = 0; i < n; i += 1) {
48         if (i == s) {
49             continue;
50         }
51         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
63             [v]);
64         u = rbdsu.find(u);
65         v = rbdsu.find(edge[v]->e.u);
66     }
67     cycles.emplace_back(edge[u], t);
68     while (heap[u] and rbdsu.find(heap[u]->e.u)
69         == rbdsu.find(u)) {
70         pop(heap[u]);
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>,
28     greater<T>>;
29 vector<i64> k_shortest_paths(int n,
30     const vector<Edge>&
31     edges,
32     int s,
33     int t,
34     int k) {
35     vector<vector<int>> g(n);
36     for (int i = 0; i < ssize(edges); i += 1) {
37         g[edges[i].u].push_back(i);
38     }
39     vector<int> par(n, -1), p;
40     vector<i64> d(n, -1);
41     minimum_heap<pair<i64, int>> pq;
42     pq.push({d[s] = 0, s});
43     while (not pq.empty()) {
44         auto [du, u] = pq.top();
45         pq.pop();
46         if (du > d[u]) {
47             continue;
48         }
49         p.push_back(u);
50         for (int i : g[u]) {
51             auto [_, v, w] = edges[i];
52             if (d[v] == -1 or d[v] > d[u] + w) {
53                 par[v] = i;
54                 pq.push({d[v] = d[u] + w, v});
55             }
56         }
57     }
58     if (d[t] == -1) {
59         return vector<i64>(k, -1);
60     }
61     vector<Node*> heap(n);
62     for (int i = 0; i < ssize(edges); i += 1) {
63         auto [u, v, w] = edges[i];
64         if (~d[u] and ~d[v] and par[v] != i) {
65             heap[v] = merge(heap[v], new Node(u, d[u] +
66                 w - d[v]));
67         }
68     }

```

```

65 }
66 for (int u : p) {
67     if (u != s) {
68         heap[u] = merge(heap[u], heap[edges[par[u]
69             ].u]);
70     }
71 }
72 minimum_heap<pair<i64, Node*>> q;
73 if (heap[t]) {
74     q.push({d[t] + heap[t]->w, heap[t]});
75 }
76 vector<i64> res = {d[t]};
77 for (int i = 1; i < k and not q.empty(); i +=
78     1) {
79     auto [w, p] = q.top();
80     q.pop();
81     res.push_back(w);
82     if (heap[p->v]) {
83         q.push({w + heap[p->v]->w, heap[p->v]});
84     }
85     for (auto c : {p->l, p->r}) {
86         if (c) {
87             q.push({w + c->w - p->w, c});
88         }
89     }
90     res.resize(k, -1);
91     return res;
92 }

```

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]
6         - pi[c]; };
7     for (int c = 0; c < n; c += 1) {
8         int r =
9             ranges::min(views::iota(0, n), {}, [&](
10                 int r) { return w[r][c]; });
11         pi[c] = w[r][c];
12         if (rm[r] == -1) {
13             rm[r] = c;
14             cm[c] = r;
15         }
16     }
17     vector<int> cols(n);
18     iota(cols.begin(), cols.end(), 0);
19     for (int r = 0; r < n; r += 1) {
20         if (rm[r] != -1) {
21             continue;
22         }
23         vector<i64> d(n);
24         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 ) {
3     int n = g.size();
4     int mark = 0;
5     vector<int> matched(n, -1), par(n, -1), book(n);
6     ;
7     auto match = [&](int s) {
8         vector<int> c(n), type(n, -1);
9         iota(c.begin(), c.end(), 0);
10        queue<int> q;
11        q.push(s);
12        type[s] = 0;
13        while (not q.empty()) {
14            int u = q.front();
15            q.pop();
16            for (int v : g[u])
17                if (type[v] == -1) {
18                    par[v] = u;
19                    type[v] = 1;
20                    int w = matched[v];
21                    if (w == -1) {
22                        [&](int u) {
23                            while (u != -1) {
24                                int v = matched[par[u]];
25                                matched[matched[u] = par[u]] = u;
26                                u = v;
27                            }
28                        }(v);
29                    }
30                    return;
31                }
32            q.push(w);
33            type[w] = 0;
34        } else if (not type[v] and c[u] != c[v]) {
35            int w = [&](int u, int v) {
36                mark += 1;
37                while (true) {
38                    if (u != -1) {
39                        if (book[u] == mark) {
40                            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 for (int i = 0; i < n; i += 1) {
70     if (matched[i] == -1) {
71         match(i);
72     }
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 +=
50                        1) {
51                        auto [_, v, f] = edges[g[u][i]];
52                        if (f and h[u] > h[v] + 1) {
53                            h[u] = h[v] + 1;
54                            cur[u] = i;
55                        }
56                    }
57                    count[h[u]] += 1;
58                    if (not(count[hi] -- == 1) and hi < n) {
59                        for (int i = 0; i < n; i += 1) {
60                            if (h[i] > hi and h[i] < n) {
61                                count[h[i]] -= 1;
62                                h[i] = n + 1;
63                            }

```

```

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)
121     -> i64 {
122     if (u == t) {
123         return limit;
124     }
125     i64 res = 0;
126     for (int& i = cur[u]; i < ssize(g[u]) and
127         limit; i += 1) {
128         int j = g[u][i];
129         auto [_, v, f] = edges[j];
130         if (level[v] == level[u] + 1 and f) {
131             if (i64 d = dfs(dfs, v, min(f, limit));
132                 d) {
133                 limit -= d;
134                 res += d;
135                 edges[j].f -= d;
136                 edges[j ^ 1].f += d;
137             }
138         }
139     }
140     return res;
141 };
142 while (bfs()) {
143     cur.assign(n, 0);
144     while (i64 f = dfs(dfs, s, numeric_limits<
145         i64>::max())) {
146         flow += f;
147     }
148 }
149 return flow;
150 };

```

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>
4         >, greater<T>>;
5     int n;
6     vector<Edge> edges;
7     vector<vector<int>>> g;
8     MinimumCostMaximumFlow(int n)
9         : n(n), g(n) {}
10    int add_edge(int u, int v, i64 f, i64 c) {
11        int i = edges.size();
12        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            ();
20        vector<i64> d, h(n);
21        vector<int> p;
22        auto dijkstra = [&]() {
23            d.assign(n, inf);
24            p.assign(n, -1);
25            minimum_heap<pair<i64, int>> q;
26            q.emplace(d[s] = 0, s);
27            while (not q.empty()) {
28                auto [du, u] = q.top();
29                q.pop();
30                if (du > d[u]) {
31                    continue;
32                }
33                for (int i : g[u]) {
34                    auto [_, v, f, c] = edges[i];
35                    if (f and d[v] > d[u] + h[u] - h[v] + c
36                        ) {
37                        p[v] = i;
38                        q.emplace(d[v] = d[u] + h[u] - h[v] +
39                            c, v);
40                    }
41                }
42            }
43            return ~p[t];
44        };
45        i64 f = 0, c = 0;
46        while (dijkstra()) {
47            for (int i = 0; i < n; i += 1) {
48                h[i] += d[i];
49            }
50            vector<int> path;
51            for (int u = t; u != s; u = edges[p[u]].u) {
52                path.push_back(p[u]);
53            }
54            i64 mf =
55                edges[ranges::min(path, {}, [&](int i)
56                    { return edges[i].f; })].f;
57            f += mf;
58            c += mf * h[t];
59            for (int i : path) {
60                edges[i].f -= mf;
61                edges[i ^ 1].f += mf;
62            }
63        }
64    }

```

```

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]
6         = find(dsu[u]); }
7     void merge(int u, int v) {
8         u = find(u);
9         v = find(v);
10        if (u != v) {
11            if (dsu[u] > dsu[v]) {
12                swap(u, v);
13            }
14            dsu[u] += dsu[v];
15            dsu[v] = u;
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(
24            dsu[u]); }
25        int time() { return ssize(stack); }
26        bool merge(int u, int v) {
27            if ((u = find(u)) == (v = find(v))) {
28                return false;
29            }
30            if (dsu[u] < dsu[v]) {
31                swap(u, v);
32            }
33            stack.emplace_back(u, dsu[u]);
34            dsu[v] += dsu[u];
35            dsu[u] = v;
36            return true;
37        }
38        void rollback(int t) {
39            while (ssize(stack) > t) {
40                auto [u, dsu_u] = stack.back();
41                stack.pop_back();
42                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                {
12                    table[i][j] = min(table[i - 1][j], table[
13                        i - 1][j + (1 << (i - 1))]);
14                }
15        }
16        int query(int l, int r) {
17            int h = bit_width(unsigned(r - l)) - 1;
18            return min(table[h][l], table[h][r - (1 << h)
19                ]);
20        }
21    };
22    struct DisjointSparseTable {
23        vector<vector<int>> table;
24        DisjointSparseTable(const vector<int>& a) {
25            int h = bit_width(a.size() - 1), n = a.size()
26            ;
27            table.resize(h, a);
28            for (int i = 0; i < h; i += 1) {
29                for (int j = 0; j + (1 << i) < n; j += (2
30                    << i)) {
31                    for (int k = j + (1 << i) - 2; k >= j; k
32                        -= 1) {
33                        table[i][k] = min(table[i][k], table[i
34                            ][k + 1]);
35                    }
36                }
37            }
38            for (int k = j + (1 << i) + 1; k < j + (2
39                << i) and k < n; k += 1) {
40                table[i][k] = min(table[i][k], table[i
41                    ][k - 1]);
42            }
43        }
44    };
45    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,
9         sizeof(Node)); }
10    Node(int v)
11        : v(v), sum(v), priority(mt()), size(1) { l
12            = r = nullptr; }
13    Node* update(Node* l, Node* r) {
14        Node* p = persistent ? new Node(*this) : this
15        ;
16        p->l = l;
17        p->r = r;
18        p->size = (l ? l->size : 0) + 1 + (r ? r->
19            size : 0);
20        p->sum = (l ? l->sum : 0) + v + (r ? r->sum :
21            0);
22        return p;
23    }
24    };
25    mt19937_64 Node::mt;
26    pair<Node*, Node*> split_by_v(Node* p, int v) {
27        if (not p) {
28            return {};
29        }
30        if (p->v < v) {
31            auto [l, r] = split_by_v(p->r, v);
32            return {p->update(p->l, l), r};
33        }
34        auto [l, r] = split_by_v(p->l, v);
35        return {l, p->update(r, p->r)};
36    }
37    pair<Node*, Node*> split_by_size(Node* p, int
38        size) {
39        if (not p) {
40            return {};
41        }
42        int l_size = p->l ? p->l->size : 0;

```

```

37     if (l_size < size) {
38         auto [l, r] = split_by_size(p->r, size -
39             l_size - 1);
40         return {p->update(p->l, l), r};
41     }
42     auto [l, r] = split_by_size(p->l, size);
43     return {l, p->update(r, p->r)};
44 }
45 Node* merge(Node* l, Node* r) {
46     if (not l or not r) {
47         return l ? r;
48     }
49     if (l->priority < r->priority) {
50         return r->update(merge(l, r->l), r->r);
51     }
52     return l->update(l->l, merge(l->r, r));
53 }

```

3.4 Lines Maximum

```

1 struct Line {
2     mutable i64 k, b, p;
3     bool operator<(const Line& rhs) const { return
4         k < rhs.k; }
5     bool operator<(const i64& x) const { return p <
6         x; }
7 };
8 struct Lines : multiset<Line, less<>> {
9     static constexpr i64 inf = numeric_limits<i64
10         >::max();
11     static i64 div(i64 a, i64 b) { return a / b -
12         ((a ^ b) < 0 and a % b); }
13     bool isect(iterator x, iterator y) {
14         if (y == end()) {
15             return x->p == inf, false;
16         }
17         if (x->k == y->k) {
18             x->p = x->b > y->b ? inf : -inf;
19         } else {
20             x->p = div(y->b - x->b, x->k - y->k);
21             return x->p >= y->p;
22         }
23     }
24     void add(i64 k, i64 b) {
25         auto z = insert({k, b, 0}), y = z++, x = y;
26         while (isect(y, z)) {
27             z = erase(z);
28         }
29         if (x != begin() and isect(--x, y)) {
30             isect(x, y = erase(y));
31         }
32     }

```

```

28     while ((y = x) != begin() and (--x)->p >= y->
29         p) {
30         isect(x, erase(y));
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
16            = [&](Node*& p, i64 tl,
17
18        if (p == nullptr) {
19            p = new Node();
20        }
21        i64 tm = midpoint(tl, tr);
22        if (tl >= l and tr <= r) {
23            if (not p->s) {
24

```

```

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 = [&](
59         i64 Node* p, i64 tl, i64 tr) {
60         if (p == nullptr) {
61             tr return;
62         }
63         i64 tm = midpoint(tl, tr);
64         Segment s {
65             i64 y = p->s.value().get(x);
66             if (not res or res.value() < y) {
67                 res = y;
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>
3         >::max() / 2;
4     i64 mv, smv, cmv, tmv;
5     bool less;
6     i64 def() { return less ? inf : -inf; }
7     i64 mmv(i64 x, i64 y) { return less ? min(x, y)
8         : max(x, y); }
9     Mv(i64 x, bool less)
10        : less(less) {
11            mv = x;
12            smv = tmv = def();
13            cmv = 1;
14        }
15    void up(const Mv& ls, const Mv& rs) {
16        mv = mmv(ls.mv, rs.mv);
17        smv = mmv(ls.mv == mv ? ls.smv : ls.mv, rs.mv
18            == mv ? rs.smv : rs.mv);
19        cmv = (ls.mv == mv ? ls.cmv : 0) + (rs.mv ==
20            mv ? rs.cmv : 0);
21    }
22    void add(i64 x) {
23        mv += x;
24        if (smv != def()) {
25            smv += x;
26        }
27        if (tmv != def()) {
28            tmv += x;
29        }
30    }
31 };
32 struct Node {
33     Mv mn, mx;
34     i64 sum, tsum;
35     Node *ls, *rs;
36     Node(i64 x = 0)
37        : sum(x), tsum(0), mn(x, true), mx(x, false)
38        {}
39     ls = rs = nullptr;
40 }
41 void up() {
42     sum = ls->sum + rs->sum;
43     mx.up(ls->mx, rs->mx);
44     mn.up(ls->mn, rs->mn);
45 }
46 void down(int tl, int tr) {
47     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 &lms = less ? mn : mx, &rms = less ? mx
        : mn;
68     if (not cmp(x, rms.mv, less)) {
69         return;
70     }
71     sum += (x - rms.mv) * rms.cmv;
72     if (lms.smv == rms.mv) {
73         lhs.smv = x;
74     }
75     if (lhs.mv == rms.mv) {
76         lhs.mv = x;
77     }
78     if (cmp(x, rms.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 &lms = less ? mn : mx, &rms = less ? mx
        : mn;
99     if (not cmp(x, rms.mv, less)) {
100         return;
101     }
102     if (tl >= l and tr <= r and cmp(rms.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         {}
10    ch.fill(nullptr);
11 }
12 int get_s() {
13     if (par == nullptr) {
14         return -1;
15     }
16     if (par->ch[0] == this) {
17         return 0;
18     }
19     if (par->ch[1] == this) {
20         return 1;

```

```

21     return -1;
22 }
23 void push_up() {
24     st = OP()(ch[0] ? ch[0]->st : E(), OP()(t,
25         ch[1] ? ch[1]->st : E()));
26 }
27 void reverse() {
28     reversed ^= 1;
29     st = REV()(st);
30 }
31 void push_down() {
32     if (reversed) {
33         swap(ch[0], ch[1]);
34         if (ch[0]) {
35             ch[0]->reverse();
36         }
37         if (ch[1]) {
38             ch[1]->reverse();
39         }
40     }
41     reversed = false;
42 }
43 void attach(int s, Node* u) {
44     if ((ch[s] = u)) {
45         u->par = this;
46     }
47     push_up();
48 }
49 void rotate() {
50     auto p = par;
51     auto pp = p->par;
52     int s = get_s();
53     int ps = p->get_s();
54     p->attach(s, ch[s ^ 1]);
55     attach(s ^ 1, p);
56     if (~ps) {
57         pp->attach(ps, this);
58     }
59     par = pp;
60 }
61 void splay() {
62     push_down();
63     while (~get_s() and ~par->get_s()) {
64         par->par->push_down();
65         par->push_down();
66         push_down();
67         (get_s() == par->get_s() ? par : this)->
68             rotate();
69     }
70     rotate();
71 }
72 if (~get_s()) {
73     par->push_down();
74     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             ]) {
8             z[i] += 1;
9         }
10        if (i + z[i] > j + z[j]) {
11            j = i;
12        }
13        return z;
14    }

```

4.2 Lyndon Factorization

```

1 vector<int> lyndon_factorization(string const& s)
2 {
3     vector<int> res = {0};
4     for (int i = 0, n = s.size(); i < n; i += 1) {
5         int j = i + 1, k = i;
6         for (; j < n and s[k] <= s[j]; j += 1) {
7             k = s[k] < s[j] ? i : k + 1;
8         }
9         while (i <= k) {
10             res.push_back(i += j - k);
11         }
12     }
13     return res;

```

4.3 Border

```

1 vector<int> fborder(const string& s) {
2     int n = s.size();
3     vector<int> res(n);
4     for (int i = 1; i < n; i += 1) {
5         int& j = res[i] = res[i - 1];
6         while (j and s[i] != s[j]) {
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 += 1) {
5     if (j + p[j] > i) {
6         p[i] = min(p[j * 2 - 1], j + p[j] - i);
7     }
8     while (i >= p[i] and i + p[i] < n and s[i - p
9         [i]] == s[i + p[i]]) {
10         p[i] += 1;
11     }
12     if (i + p[i] > j + p[j]) {
13         j = i;
14     }
15     return p;
16 }

```

4.5 Suffix Array

```

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

```

```

30         previous[p[i] + m] ==
31             previous[p[i - 1] +
32                 m]
33             ? rank[p[i - 1]]
34             : k++;
35     }
36     vector<int> lcp(n);
37     k = 0;
38     for (int i = 0; i < n; i += 1) {
39         if (rank[i]) {
40             k = max(k - 1, 0);
41             int j = p[rank[i] - 1];
42             while (i + k < n and j + k < n and s[i + k]
43                 == s[j + k]) {
44                 k += 1;
45             }
46             lcp[rank[i]] = k;
47         }
48     }
49     return {p, lcp};

```

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     while (not q.empty()) {
30         int u = q.front();
31         q.pop();
32         for (int i = 0; i < sigma; i += 1) {
33             if (at(u).next[i]) {
34                 at(at(u).next[i]).link = at(at(u).link)
35                     .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))
30             .next[s[i]];
31         back().sum = at(back().link).sum + 1;
32         at(cur).next[s[i]] = now;
33     }
34     last = at(cur).next[s[i]];
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,
16         y - rhs.y}; }
17     Gauss operator*(Gauss rhs) {
18         return {x * rhs.x - y * rhs.y, x * rhs.y + y
19             * rhs.x};
20     }
21     Gauss operator/(Gauss rhs) {
22         auto [x, y] = operator*(~rhs);
23         return {div_round(x, rhs.norm()), div_round(y,
24             rhs.norm())};
25     }
26     Gauss operator%(Gauss rhs) { return operator-(
27         rhs*(operator/(rhs))); }
28 };

```

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>
16        b) {
17        return pair((a.first * b.first + a.second * b
18            .second % p * w) % p,
19            (a.first * b.second + a.second *
20                b.first) % p);
21    };
22    pair<i64, i64> a = {x, 1}, res = {1, 0};
23    for (i64 r = (p + 1) >> 1; r; r >>= 1, a = mul(
24        a, a)) {
25        if (r & 1) {
26            res = mul(res, a);
27        }
28    }
29    return res.first;
30 }

```

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;
10         res += 1) {
11         if (y % d) {
12             return -1;
13         }
14         n /= d;
15         y /= d;
16         k = k * (x / d) % n;
17     }
18 }

```

```

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 {
24     mp[y * px % n] = i;
25 }
26 i64 ppx = k * px % n;
27 for (int i = 1; i <= m; i += 1, ppx = ppx * px
28     % n) {
29     if (mp.count(ppx)) {
30         return res + i * m - mp[ppx];
31     }
32 }
33 return -1;
34 }

```

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,
12     i64 b0, i64 a1, i64 b1) {
13     auto [d, x, y] = exgcd(a0, a1);
14     if ((b1 - b0) % d) {
15         return {};
16     }
17     i64 a = a0 / d * a1, b = (i128)(b1 - b0) / d *
18         x % (a1 / d);
19     if (b < 0) {
20         b += a1 / d;
21     }
22     b = (i128)(a0 * b + b0) % a;
23     if (b < 0) {
24         b += a;
25     }
26     return {{a, b}};
27 }

```

5.4 Miller Rabin

```

1 bool miller_rabin(i64 n) {
2     static constexpr array<int, 9> p = {2, 3, 5, 7,
3         11, 13, 17, 19, 23};
4     if (n == 1) {
5         return false;
6     }
7     if (n == 2) {
8         return true;
9     }
10    if (not(n % 2)) {
11        return false;
12    }
13    int r = countr_zero(u64(n - 1));
14    i64 d = (n - 1) >> r;
15    for (int pi : p) {
16        if (pi >= n) {
17            break;
18        }
19        i64 x = power(pi, d, n);
20        if (x == 1 or x == n - 1) {
21            continue;
22        }
23        for (int j = 1; j < r; j += 1) {
24            x = (i128)x * x % n;
25            if (x == n - 1) {
26                break;
27            }
28        }
29        if (x != n - 1) {
30            return false;
31        }
32    }
33    return true;
34 }

```

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 =
16         uid(mt); d == 1;
17         k <= 1, y = x, s = 1) {
18         for (int i = 1; i <= k; i += 1) {
19             x = ((i128)x * x + c) % n;
20             s = (i128)s * abs(x - y) % n;
21             if (not(i % 127) or i == k) {
22                 d = gcd(s, n);
23                 if (d != 1) {
24                     break;
25                 }
26             }
27         }
28     }
29     rho(d);
30     rho(n / d);
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
2     = 1, u64 p = 1, u64 q = 1) {
3     if (a == 0) {
4         return b;
5     }
6     if (c % 2) {
7         if (b >= a) {
8             u64 t = (m - b + a - 1) / a;
9             u64 d = (t - 1) * p + q;
10            if (n <= d) {
11                return b;
12            }
13            n -= d;
14            b += a * t - m;
15        }
16        b = a - 1 - b;
17    } else {
18        if (b < m - a) {
19            u64 t = (m - b - 1) / a;
20            u64 d = t * p;
21            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
30     - 1) * p + q, d * p + q);
31 return c % 2 ? m - 1 - res : a - 1 - res;
32 }

```

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
20        _b = 1)
21        : p(p), a(_a), b(_b) {
22        for (auto [c, d] : p | views::reverse) {
23            if (d == 'R') {
24                a += c * b;
25            } else {
26                b += c * a;
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 ^
13                        x) | (i & (x - 1))];
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 +=
22                    1) {
23                    if ((y >> j) % 2) {
24                        res ^= mem[i][j];
25                    }
26                }
27            }
28        }
29        return res;
30    };

```

6 Numerical

6.1 Golden Search

```

1 template <int step>
2 f64 golden_search(function<f64(f64)> f, f64 l,
3     f64 r) {
4     f64 ml = (numbers::phi - 1) * l + (2 - numbers
5         ::phi) * r;
6     f64 mr = l + r - ml;
7     f64 fml = f(ml), fmr = f(mr);
8     for (int i = 0; i < step; i += 1)
9         if (fml > fmr) {
10            l = ml;
11            ml = mr;
12            fml = fmr;
13            fmr = f(mr = (numbers::phi - 1) * r + (2 -
14                numbers::phi) * l);
15        } else {
16            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 / 2);
14 }
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]
43                                     > base[k])) {
44                 k = j;
45             }
46         }
47         if (k == -1) {
48             return false;
49         }
50         pivot(i, k);
51     }
52     return true;
53 }
54 bool bounded() {
55     while (true) {
56         int i = ranges::max_element(c) - c.begin();
57         if (c[i] < eps) {
58             break;
59         }
60         int k = -1;
61         for (int j = 0; j < m; j += 1) {
62             if (a[j][i] > eps) {
63                 if (k == -1) {
64                     k = j;
65                 } else {
66                     f64 d = b[j] * a[k][i] - b[k] * a[j][i];
67                     if (d < -eps or (d < eps and base[j]
68                                     > base[k])) {
69                         k = j;
70                     }
71                 }
72             }
73         }
74     }
75 }

```

```

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

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

6.5 Double Integral

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

7 Convolution

7.1 Fast Fourier Transform on \mathbb{C}

```

1 void fft(vector<complex<f64>>& a, bool inverse) {
2     int n = a.size();
3     vector<int> r(n);
4     for (int i = 0; i < n; i += 1) {
5         r[i] = r[i / 2] / 2 | (i % 2 ? n / 2 : 0);
6     }
7     for (int i = 0; i < n; i += 1) {
8         if (i < r[i]) {
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     {
18         auto &x = a[i + j + m], &y = a[i + j], t
19         = w * x;
20         tie(x, y) = pair(y - t, y + t);
21     }
22 }
23 if (inverse) {
24     for (auto& ai : a) {
25         ai /= n;
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) :
14            g, (mod - 1) / m / 2);
15        for (int i = 0; i < n; i += m * 2) {
16            i64 w = 1;
17            for (int j = 0; j < m; j += 1, w = w * wn %
18                mod) {
19                auto &x = a[i + j + m], &y = a[i + j], t
20                = w * x % mod;
21                tie(x, y) = pair((y + mod - t) % mod, (y
22                    + t) % mod);
23            }
24        }
25    }
26    if (inverse) {
27        i64 inv = power(n, mod - 2);
28        for (auto& ai : a) {
29            ai = ai * inv % mod;
30        }
31    }
32 }

```

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

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

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) ==
22         0 and sign(y - p.y) == 0; }
22     int arg() { return y < 0 or (y == 0 and x > 0)
23         ? -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
40             sign((p - a).dot(p - b)) <= 0;
41     }
42     int left(P<T> p) { return sign(v().cross(p - a))
43         > 0; }
44     optional<pair<T, T>> intersection(L l) {
45         auto y = v().cross(l.v());
46         if (sign(y) == 0) {

```

<pre> 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) 76 >= 0) { 77 res += 1; 78 } 79 if (sign(a.y - p.y) >= 0 and sign(b.y - p.y) 80 < 0) { 81 res -= 1; 82 } 83 } 84 return res; 85 } 86 G convex() { 87 ranges::sort(g, {}, [&](P<T> p) { return pair 88 (p.x, p.y); }); 89 vector<P<T>> h; 90 for (auto p : g) { 91 while (ssize(h) >= 2 and 92 sign((h.back() - h.end()[-2]).cross(93 p - h.back())) <= 0) { </pre>	<pre> 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(98 p - h.back())) <= 0) { 99 h.pop_back(); 100 } 101 h.push_back(p); 102 } 103 h.pop_back(); 104 return G(h); 105 } 106 // Following function are valid only for convex 107 . 108 T diameter2() { 109 int n = g.size(); 110 T res = 0; 111 for (int i = 0, j = 1; i < n; i += 1) { 112 auto a = g[i], b = g[(i + 1) % n]; 113 while (sign((b - a).cross(g[(j + 1) % n] - 114 g[j])) > 0) { 115 j = (j + 1) % n; 116 } 117 res = max(res, (a - g[j]).len2()); 118 res = max(res, (a - g[j]).len2()); 119 } 120 return res; 121 } 122 optional<bool> contains(P<T> p) { 123 if (g[0] == p) { 124 return {}; 125 } 126 if (g.size() == 1) { 127 return false; 128 } 129 if (L(g[0], g[1]).contains(p)) { 130 return {}; 131 } 132 if (L(g[0], g[1]).left(p) <= 0) { 133 return false; 134 } 135 if (L(g[0], g.back()).left(p) > 0) { 136 return false; 137 } 138 int i = *ranges::partition_point(views::iota 139 (2, ssize(g)), [&](int i) { 140 return sign((p - g[0]).cross(g[i] - g[0])) 141 <= 0; </pre>	<pre> 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] - 148 g[i])) >= 0; 149 }; 150 P<T> f0 = f(g[0]); 151 bool check0 = check(0); 152 if (not check0 and check(n - 1)) { 153 return 0; 154 } 155 return *ranges::partition_point(views::iota 156 (0, n), [&](int i) -> bool { 157 if (i == 0) { 158 return true; 159 } 160 bool checki = check(i); 161 int t = sign(f0.cross(g[i] - g[0])); 162 if (i == 1 and checki == check0 and t == 0) { 163 return true; 164 } 165 return checki ^ (checki == check0 and t <= 166 0); 167 }); 168 } 169 pair<int, int> tan(P<T> p) { 170 return {most([&](P<T> x) { return x - p; }), 171 most([&](P<T> x) { return p - x; })}; 172 } 173 pair<int, int> tan(L<T> l) { 174 return {most([&](P<T> _) { return l.v(); }), 175 most([&](P<T> _) { return -l.v(); })}; 176 } 177 template <typename T> 178 vector<L<T>> half(vector<L<T>> ls, T bound) { 179 // Ranges: bound ^ 6 180 auto check = []<T>(a, L<T> b, L<T> c) { 181 auto [x, y] = b.intersection(c).value(); 182 a = L(a.a * y, a.b * y); 183 return a.left(b.a * y + b.v() * x) < 0; 184 }; 185 ls.emplace_back(P(-bound, (T)0), P(-bound, -(T) 186 1)); 187 ls.emplace_back(P((T)0, -bound), P((T)1, -bound 188)); 189 ls.emplace_back(P(bound, (T)0), P(bound, (T)1)); 190 } </pre>
--	--	---

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