

Team Reference Document

the-tourist

2024 年 5 月 11 日

目录

1 data

1.1 bstnode.cpp

```
1 class node {
2     public:
3         int id;
4         node* l;
5         node* r;
6         node* p;
7         bool rev;
8         int sz;
9         // declare extra variables:
10
11
12     node(int _id) {
13         id = _id;
14         l = r = p = nullptr;
15         rev = false;
16         sz = 1;
17         // init extra variables:
18
19     }
20
21     // push everything else:
22     void push_stuff() {
23
24     }
25
26     void unsafe_reverse() {
27         push_stuff(); // !! edu 112
28         rev ^= 1;
29         swap(l, r);
30         pull();
31     }
32
33     // apply changes:
34     void unsafe_apply() {
35
```

```
36     }
37
38     void push() {
39         if (rev) {
40             if (l != nullptr) {
41                 l->unsafe_reverse();
42             }
43             if (r != nullptr) {
44                 r->unsafe_reverse();
45             }
46             rev = 0;
47         }
48         push_stuff();
49     }
50
51     void pull() {
52         sz = 1;
53         // now init from self:
54
55         if (l != nullptr) {
56             l->p = this;
57             sz += l->sz;
58             // now pull from l:
59
60         }
61         if (r != nullptr) {
62             r->p = this;
63             sz += r->sz;
64             // now pull from r:
65
66         }
67     }
68 };
69
70 void debug_node(node* v, string pref = "") {
71     #ifdef LOCAL
72         if (v != nullptr) {
73             debug_node(v->r, pref + "└");
74             cerr << pref << "-" << "└" << v->id << '\n';
75             debug_node(v->l, pref + "└");
76         } else {
77             cerr << pref << "-" << "└" << "nullptr" << '\n';

```

```

78     }
79     #endif
80 }

```

1.2 dsu.cpp

```

1  class dsu {
2  public:
3      vector<int> p;
4      int n;
5
6      dsu(int _n) : n(_n) {
7          p.resize(n);
8          iota(p.begin(), p.end(), 0);
9      }
10
11     inline int get(int x) {
12         return (x == p[x] ? x : (p[x] = get(p[x])));
13     }
14
15     inline bool unite(int x, int y) {
16         x = get(x);
17         y = get(y);
18         if (x != y) {
19             p[x] = y;
20             return true;
21         }
22         return false;
23     }
24 };

```

1.3 fenwick.cpp

```

1  template <typename T>
2  class fenwick {
3  public:
4      vector<T> fenw;
5      int n;
6
7      fenwick(int _n) : n(_n) {
8          fenw.resize(n);

```

```

9      }
10
11     void modify(int x, T v) {
12         while (x < n) {
13             fenw[x] += v;
14             x |= (x + 1);
15         }
16     }
17
18     T get(int x) {
19         T v{};
20         while (x >= 0) {
21             v += fenw[x];
22             x = (x & (x + 1)) - 1;
23         }
24         return v;
25     }
26 };
27
28 struct node {
29     int a = ...; // don't forget to set default value
30
31     inline void operator += (node &other) {
32         ...
33     }
34 };

```

1.4 fenwick2d.cpp

```

1  template <typename T>
2  class fenwick2d {
3  public:
4      vector<vector<T>> fenw;
5      int n, m;
6
7      fenwick2d(int _n, int _m) : n(_n), m(_m) {
8          fenw.resize(n);
9          for (int i = 0; i < n; i++) {
10             fenw[i].resize(m);
11         }
12     }

```

```

13
14 inline void modify(int i, int j, T v) {
15     int x = i;
16     while (x < n) {
17         int y = j;
18         while (y < m) {
19             fenw[x][y] += v;
20             y |= (y + 1);
21         }
22         x |= (x + 1);
23     }
24 }
25
26 inline T get(int i, int j) {
27     T v{};
28     int x = i;
29     while (x >= 0) {
30         int y = j;
31         while (y >= 0) {
32             v += fenw[x][y];
33             y = (y & (y + 1)) - 1;
34         }
35         x = (x & (x + 1)) - 1;
36     }
37     return v;
38 }
39 };
40
41 struct node {
42     int a = ...; // don't forget to set default value
43
44     inline void operator += (node &other) {
45         ...
46     }
47 };

```

1.5 linkcut.cpp

```

1 template <bool rooted>
2 class link_cut_tree {
3 public:

```

```

4     int n;
5     vector<node*> nodes;
6
7     link_cut_tree(int _n) : n(_n) {
8         nodes.resize(n);
9         for (int i = 0; i < n; i++) {
10             nodes[i] = new node(i);
11         }
12     }
13
14     int add_node() {
15         int id = (int) nodes.size();
16         nodes.push_back(new node(id));
17         return id;
18     }
19
20     void expose(node* v) {
21         node* r = nullptr;
22         node* u = v;
23         while (u != nullptr) {
24             splay(u);
25             u->push();
26             u->r = r;
27             u->pull();
28             r = u;
29             u = u->p;
30         }
31         splay(v);
32         assert(v->p == nullptr);
33     }
34
35     int get_root(int i) {
36         node* v = nodes[i];
37         expose(v);
38         return get_leftmost(v)->id;
39     }
40
41     bool link(int i, int j) { // for rooted: (x, parent[x])
42         if (i == j) {
43             return false;
44         }
45         node* v = nodes[i];

```

```

46     node* u = nodes[j];
47     if (rooted) {
48         splay(v);
49         if (v->p != nullptr || v->l != nullptr) {
50             return false; // not a root
51         }
52     } else {
53         make_root(i);
54     }
55     expose(u);
56     if (v->p != nullptr) {
57         return false;
58     }
59     v->p = u;
60     return true;
61 }
62
63 bool cut(int i, int j) { // for rooted: (x, parent[x])
64     if (i == j) {
65         return false;
66     }
67     node* v = nodes[i];
68     node* u = nodes[j];
69     expose(u);
70     splay(v);
71     if (v->p != u) {
72         if (rooted) {
73             return false;
74         }
75         swap(u, v);
76         expose(u);
77         splay(v);
78         if (v->p != u) {
79             return false;
80         }
81     }
82     v->p = nullptr;
83     return true;
84 }
85
86 bool cut(int i) { // only for rooted
87     assert(rooted);

```

```

88     node* v = nodes[i];
89     expose(v);
90     v->push();
91     if (v->l == nullptr) {
92         return false; // already a root
93     }
94     v->l->p = nullptr;
95     v->l = nullptr;
96     v->pull();
97     return true;
98 }
99
100 bool connected(int i, int j) {
101     if (i == j) {
102         return true;
103     }
104     node* v = nodes[i];
105     node* u = nodes[j];
106     expose(v);
107     expose(u);
108     return v->p != nullptr;
109 }
110
111 int lca(int i, int j) {
112     if (i == j) {
113         return i;
114     }
115     node* v = nodes[i];
116     node* u = nodes[j];
117     expose(v);
118     expose(u);
119     if (v->p == nullptr) {
120         return -1;
121     }
122     splay(v);
123     if (v->p == nullptr) {
124         return v->id;
125     }
126     return v->p->id;
127 }
128
129 bool is_ancestor(int i, int j) {

```

```

130     if (i == j) {
131         return true;
132     }
133     node* v = nodes[i];
134     node* u = nodes[j];
135     expose(u);
136     splay(v);
137     return v->p == nullptr && u->p != nullptr;
138 }
139
140 void make_root(int i) {
141     assert(!rooted);
142     node* v = nodes[i];
143     expose(v);
144     reverse(v);
145 }
146
147 node* get_path_from_root(int i) {
148     node* v = nodes[i];
149     expose(v);
150     return v;
151 }
152
153 template <typename... T>
154 void apply(int i, T... args) {
155     node* v = nodes[i];
156     splay_tree::apply(v, args...);
157 }
158 };

```

1.6 pbds.cpp

```

1 #include <ext/pb_ds/assoc_container.hpp>
2
3 using namespace __gnu_pbds;
4
5 typedef int tp;
6 typedef tree<tp,null_type,less<tp>,rb_tree_tag,
7     tree_order_statistics_node_update> pbds;
8 // tp a;
9 // T.insert(a), T.erase(a), T.size()

```

```

9 // T.order_of_key(a) -- number of elements strictly less than a
10 // *T.find_by_order(k) -- k-th element in increasing order

```

1.7 segtree.cpp

```

1 class segtree {
2     public:
3         struct node {
4             // don't forget to set default value (used for leaves)
5             // not necessarily neutral element!
6             ... a = ...;
7
8             void apply(int l, int r, ... v) {
9                 ...
10            }
11        };
12
13        node unite(const node &a, const node &b) const {
14            node res;
15            ...
16            return res;
17        }
18
19        inline void push(int x, int l, int r) {
20            int y = (l + r) >> 1;
21            int z = x + ((y - l + 1) << 1);
22            // push from x into (x + 1) and z
23            ...
24            /*
25             if (tree[x].add != 0) {
26                 tree[x + 1].apply(l, y, tree[x].add);
27                 tree[z].apply(y + 1, r, tree[x].add);
28                 tree[x].add = 0;
29             }
30            */
31        }
32
33        inline void pull(int x, int z) {
34            tree[x] = unite(tree[x + 1], tree[z]);
35        }
36    };

```

```

37     int n;
38     vector<node> tree;
39
40     void build(int x, int l, int r) {
41         if (l == r) {
42             return;
43         }
44         int y = (l + r) >> 1;
45         int z = x + ((y - l + 1) << 1);
46         build(x + 1, l, y);
47         build(z, y + 1, r);
48         pull(x, z);
49     }
50
51     template <typename M>
52     void build(int x, int l, int r, const vector<M> &v) {
53         if (l == r) {
54             tree[x].apply(l, r, v[l]);
55             return;
56         }
57         int y = (l + r) >> 1;
58         int z = x + ((y - l + 1) << 1);
59         build(x + 1, l, y, v);
60         build(z, y + 1, r, v);
61         pull(x, z);
62     }
63
64     node get(int x, int l, int r, int ll, int rr) {
65         if (ll <= l && r <= rr) {
66             return tree[x];
67         }
68         int y = (l + r) >> 1;
69         int z = x + ((y - l + 1) << 1);
70         push(x, l, r);
71         node res{};
72         if (rr <= y) {
73             res = get(x + 1, l, y, ll, rr);
74         } else {
75             if (ll > y) {
76                 res = get(z, y + 1, r, ll, rr);
77             } else {
78                 res = unite(get(x + 1, l, y, ll, rr), get(z, y + 1, r, ll, rr));

```

```

79         }
80     }
81     pull(x, z);
82     return res;
83 }
84
85     template <typename... M>
86     void modify(int x, int l, int r, int ll, int rr, const M&... v) {
87         if (ll <= l && r <= rr) {
88             tree[x].apply(l, r, v...);
89             return;
90         }
91         int y = (l + r) >> 1;
92         int z = x + ((y - l + 1) << 1);
93         push(x, l, r);
94         if (ll <= y) {
95             modify(x + 1, l, y, ll, rr, v...);
96         }
97         if (rr > y) {
98             modify(z, y + 1, r, ll, rr, v...);
99         }
100         pull(x, z);
101     }
102
103     int find_first_knowingly(int x, int l, int r, const function<bool(const
104         node&> &f) {
105         if (l == r) {
106             return l;
107         }
108         push(x, l, r);
109         int y = (l + r) >> 1;
110         int z = x + ((y - l + 1) << 1);
111         int res;
112         if (f(tree[x + 1])) {
113             res = find_first_knowingly(x + 1, l, y, f);
114         } else {
115             res = find_first_knowingly(z, y + 1, r, f);
116         }
117         pull(x, z);
118         return res;
119     }

```

```

120 int find_first(int x, int l, int r, int ll, int rr, const function<bool(
    const node&>> &f) {
121     if (ll <= l && r <= rr) {
122         if (!f(tree[x])) {
123             return -1;
124         }
125         return find_first_knowingly(x, l, r, f);
126     }
127     push(x, l, r);
128     int y = (l + r) >> 1;
129     int z = x + ((y - l + 1) << 1);
130     int res = -1;
131     if (ll <= y) {
132         res = find_first(x + 1, l, y, ll, rr, f);
133     }
134     if (rr > y && res == -1) {
135         res = find_first(z, y + 1, r, ll, rr, f);
136     }
137     pull(x, z);
138     return res;
139 }
140
141 int find_last_knowingly(int x, int l, int r, const function<bool(const
    node&>> &f) {
142     if (l == r) {
143         return l;
144     }
145     push(x, l, r);
146     int y = (l + r) >> 1;
147     int z = x + ((y - l + 1) << 1);
148     int res;
149     if (f(tree[z])) {
150         res = find_last_knowingly(z, y + 1, r, f);
151     } else {
152         res = find_last_knowingly(x + 1, l, y, f);
153     }
154     pull(x, z);
155     return res;
156 }
157
158 int find_last(int x, int l, int r, int ll, int rr, const function<bool(
    const node&>> &f) {

```

```

159     if (ll <= l && r <= rr) {
160         if (!f(tree[x])) {
161             return -1;
162         }
163         return find_last_knowingly(x, l, r, f);
164     }
165     push(x, l, r);
166     int y = (l + r) >> 1;
167     int z = x + ((y - l + 1) << 1);
168     int res = -1;
169     if (rr > y) {
170         res = find_last(z, y + 1, r, ll, rr, f);
171     }
172     if (ll <= y && res == -1) {
173         res = find_last(x + 1, l, y, ll, rr, f);
174     }
175     pull(x, z);
176     return res;
177 }
178
179 segtree(int _n) : n(_n) {
180     assert(n > 0);
181     tree.resize(2 * n - 1);
182     build(0, 0, n - 1);
183 }
184
185 template <typename M>
186 segtree(const vector<M> &v) {
187     n = v.size();
188     assert(n > 0);
189     tree.resize(2 * n - 1);
190     build(0, 0, n - 1, v);
191 }
192
193 node get(int ll, int rr) {
194     assert(0 <= ll && ll <= rr && rr <= n - 1);
195     return get(0, 0, n - 1, ll, rr);
196 }
197
198 node get(int p) {
199     assert(0 <= p && p <= n - 1);
200     return get(0, 0, n - 1, p, p);

```



```

201     }
202
203     template <typename... M>
204     void modify(int ll, int rr, const M&... v) {
205         assert(0 <= ll && ll <= rr && rr <= n - 1);
206         modify(0, 0, n - 1, ll, rr, v...);
207     }
208
209     // find_first and find_last call all FALSE elements
210     // to the left (right) of the sought position exactly once
211
212     int find_first(int ll, int rr, const function<bool(const node&)> &f) {
213         assert(0 <= ll && ll <= rr && rr <= n - 1);
214         return find_first(0, 0, n - 1, ll, rr, f);
215     }
216
217     int find_last(int ll, int rr, const function<bool(const node&)> &f) {
218         assert(0 <= ll && ll <= rr && rr <= n - 1);
219         return find_last(0, 0, n - 1, ll, rr, f);
220     }
221 };

```

1.8 sparsetable.cpp

```

1  template <typename T, typename F>
2  class SparseTable {
3  public:
4      int n;
5      vector<vector<T>> mat;
6      F func;
7
8      SparseTable(const vector<T>& a, const F& f) : func(f) {
9          n = static_cast<int>(a.size());
10         int max_log = 32 - __builtin_clz(n);
11         mat.resize(max_log);
12         mat[0] = a;
13         for (int j = 1; j < max_log; j++) {
14             mat[j].resize(n - (1 << j) + 1);
15             for (int i = 0; i <= n - (1 << j); i++) {
16                 mat[j][i] = func(mat[j - 1][i], mat[j - 1][i + (1 << (j - 1))]);
17             }

```

```

18     }
19 }
20
21 T get(int from, int to) const {
22     assert(0 <= from && from <= to && to <= n - 1);
23     int lg = 32 - __builtin_clz(to - from + 1) - 1;
24     return func(mat[lg][from], mat[lg][to - (1 << lg) + 1]);
25 }
26 };

```

1.9 splay.cpp

```

1  namespace splay_tree {
2
3  bool is_bst_root(node* v) {
4      if (v == nullptr) {
5          return false;
6      }
7      return (v->p == nullptr || (v->p->l != v && v->p->r != v));
8  }
9
10 void rotate(node* v) {
11     node* u = v->p;
12     assert(u != nullptr);
13     u->push();
14     v->push();
15     v->p = u->p;
16     if (v->p != nullptr) {
17         if (v->p->l == u) {
18             v->p->l = v;
19         }
20         if (v->p->r == u) {
21             v->p->r = v;
22         }
23     }
24     if (v == u->l) {
25         u->l = v->r;
26         v->r = u;
27     } else {
28         u->r = v->l;
29         v->l = u;

```

```

30     }
31     u->pull();
32     v->pull();
33 }
34
35 void splay(node* v) {
36     if (v == nullptr) {
37         return;
38     }
39     while (!is_bst_root(v)) {
40         node* u = v->p;
41         if (!is_bst_root(u)) {
42             if ((u->l == v) ^ (u->p->l == u)) {
43                 rotate(v);
44             } else {
45                 rotate(u);
46             }
47         }
48         rotate(v);
49     }
50 }
51
52 pair<node*, int> find(node* v, const function<int(node*)> &go_to) {
53     // go_to returns: 0 -- found; -1 -- go left; 1 -- go right
54     // find returns the last vertex on the descent and its go_to
55     if (v == nullptr) {
56         return {nullptr, 0};
57     }
58     splay(v);
59     int dir;
60     while (true) {
61         v->push();
62         dir = go_to(v);
63         if (dir == 0) {
64             break;
65         }
66         node* u = (dir == -1 ? v->l : v->r);
67         if (u == nullptr) {
68             break;
69         }
70         v = u;
71     }

```

```

72     splay(v);
73     return {v, dir};
74 }
75
76 node* get_leftmost(node* v) {
77     return find(v, [&](node*) { return -1; }).first;
78 }
79
80 node* get_rightmost(node* v) {
81     return find(v, [&](node*) { return 1; }).first;
82 }
83
84 node* get_kth(node* v, int k) { // 0-indexed
85     pair<node*, int> p = find(v, [&](node* u) {
86         if (u->l != nullptr) {
87             if (u->l->sz > k) {
88                 return -1;
89             }
90             k -= u->l->sz;
91         }
92         if (k == 0) {
93             return 0;
94         }
95         k--;
96         return 1;
97     });
98     return (p.second == 0 ? p.first : nullptr);
99 }
100
101 int get_position(node* v) { // 0-indexed
102     splay(v);
103     return (v->l != nullptr ? v->l->sz : 0);
104 }
105
106 node* get_bst_root(node* v) {
107     splay(v);
108     return v;
109 }
110
111 pair<node*, node*> split(node* v, const function<bool(node*)> &is_right) {
112     if (v == nullptr) {
113         return {nullptr, nullptr};

```

```

114     }
115     pair<node*, int> p = find(v, [&](node* u) { return is_right(u) ? -1 : 1;
116         });
117     v = p.first;
118     v->push();
119     if (p.second == -1) {
120         node* u = v->l;
121         if (u == nullptr) {
122             return {nullptr, v};
123         }
124         v->l = nullptr;
125         u->p = v->p;
126         u = get_rightmost(u);
127         v->p = u;
128         v->pull();
129         return {u, v};
130     } else {
131         node* u = v->r;
132         if (u == nullptr) {
133             return {v, nullptr};
134         }
135         v->r = nullptr;
136         v->pull();
137         return {v, u};
138     }
139
140     pair<node*, node*> split_leftmost_k(node* v, int k) {
141         return split(v, [&](node* u) {
142             int left_and_me = (u->l != nullptr ? u->l->sz : 0) + 1;
143             if (k >= left_and_me) {
144                 k -= left_and_me;
145                 return false;
146             }
147             return true;
148         });
149     }
150
151     node* merge(node* v, node* u) {
152         if (v == nullptr) {
153             return u;
154         }

```

```

155         if (u == nullptr) {
156             return v;
157         }
158         v = get_rightmost(v);
159         assert(v->r == nullptr);
160         splay(u);
161         v->push();
162         v->r = u;
163         v->pull();
164         return v;
165     }
166
167     int count_left(node* v, const function<bool(node*)> &is_right) {
168         if (v == nullptr) {
169             return 0;
170         }
171         pair<node*, int> p = find(v, [&](node* u) { return is_right(u) ? -1 : 1;
172             });
173         node* u = p.first;
174         return (u->l != nullptr ? u->l->sz : 0) + (p.second == 1);
175     }
176
177     node* add(node* r, node* v, const function<bool(node*)> &go_left) {
178         pair<node*, node*> p = split(r, go_left);
179         return merge(p.first, merge(v, p.second));
180     }
181
182     node* remove(node* v) { // returns the new root
183         splay(v);
184         v->push();
185         node* x = v->l;
186         node* y = v->r;
187         v->l = v->r = nullptr;
188         node* z = merge(x, y);
189         if (z != nullptr) {
190             z->p = v->p;
191         }
192         v->p = nullptr;
193         v->push();
194         v->pull(); // now v might be reusable...
195         return z;

```

```

196
197 node* next(node* v) {
198     splay(v);
199     v->push();
200     if (v->r == nullptr) {
201         return nullptr;
202     }
203     v = v->r;
204     while (v->l != nullptr) {
205         v->push();
206         v = v->l;
207     }
208     splay(v);
209     return v;
210 }
211
212 node* prev(node* v) {
213     splay(v);
214     v->push();
215     if (v->l == nullptr) {
216         return nullptr;
217     }
218     v = v->l;
219     while (v->r != nullptr) {
220         v->push();
221         v = v->r;
222     }
223     splay(v);
224     return v;
225 }
226
227 int get_size(node* v) {
228     splay(v);
229     return (v != nullptr ? v->sz : 0);
230 }
231
232 template<typename... T>
233 void do_apply(node* v, T... args) {
234     splay(v);
235     v->unsafe_apply(args...);
236 }
237

```

```

238 void reverse(node* v) {
239     splay(v);
240     v->unsafe_reverse();
241 }
242
243 } // namespace splay_tree
244
245 using namespace splay_tree;

```

1.10 treap.cpp

```

1 namespace treap {
2
3 pair<node*, int> find(node* v, const function<int(node*)> &go_to) {
4     // go_to returns: 0 -- found; -1 -- go left; 1 -- go right
5     // find returns the last vertex on the descent and its go_to
6     if (v == nullptr) {
7         return {nullptr, 0};
8     }
9     int dir;
10    while (true) {
11        v->push();
12        dir = go_to(v);
13        if (dir == 0) {
14            break;
15        }
16        node* u = (dir == -1 ? v->l : v->r);
17        if (u == nullptr) {
18            break;
19        }
20        v = u;
21    }
22    return {v, dir};
23 }
24
25 node* get_leftmost(node* v) {
26     return find(v, [&](node*) { return -1; }).first;
27 }
28
29 node* get_rightmost(node* v) {
30     return find(v, [&](node*) { return 1; }).first;

```

```

31 }
32
33 node* get_kth(node* v, int k) { // 0-indexed
34     pair<node*, int> p = find(v, [&](node* u) {
35         if (u->l != nullptr) {
36             if (u->l->sz > k) {
37                 return -1;
38             }
39             k -= u->l->sz;
40         }
41         if (k == 0) {
42             return 0;
43         }
44         k--;
45         return 1;
46     });
47     return (p.second == 0 ? p.first : nullptr);
48 }
49
50 int get_position(node* v) { // 0-indexed
51     int k = (v->l != nullptr ? v->l->sz : 0);
52     while (v->p != nullptr) {
53         if (v == v->p->r) {
54             k++;
55             if (v->p->l != nullptr) {
56                 k += v->p->l->sz;
57             }
58         }
59         v = v->p;
60     }
61     return k;
62 }
63
64 node* get_bst_root(node* v) {
65     while (v->p != nullptr) {
66         v = v->p;
67     }
68     return v;
69 }
70
71 pair<node*, node*> split(node* v, const function<bool(node*)> &is_right) {
72     if (v == nullptr) {

```

```

73         return {nullptr, nullptr};
74     }
75     v->push();
76     if (is_right(v)) {
77         pair<node*, node*> p = split(v->l, is_right);
78         if (p.first != nullptr) {
79             p.first->p = nullptr;
80         }
81         v->l = p.second;
82         v->pull();
83         return {p.first, v};
84     } else {
85         pair<node*, node*> p = split(v->r, is_right);
86         v->r = p.first;
87         if (p.second != nullptr) {
88             p.second->p = nullptr;
89         }
90         v->pull();
91         return {v, p.second};
92     }
93 }
94
95 pair<node*, node*> split_leftmost_k(node* v, int k) {
96     return split(v, [&](node* u) {
97         int left_and_me = (u->l != nullptr ? u->l->sz : 0) + 1;
98         if (k >= left_and_me) {
99             k -= left_and_me;
100             return false;
101         }
102         return true;
103     });
104 }
105
106 node* merge(node* v, node* u) {
107     if (v == nullptr) {
108         return u;
109     }
110     if (u == nullptr) {
111         return v;
112     }
113     if (v->P > u->P) {
114         // if (rng() % (v->sz + u->sz) < (unsigned int) v->sz) {

```

```

115     v->push();
116     v->r = merge(v->r, u);
117     v->pull();
118     return v;
119 } else {
120     u->push();
121     u->l = merge(v, u->l);
122     u->pull();
123     return u;
124 }
125 }
126
127 int count_left(node* v, const function<bool(node*)> &is_right) {
128     if (v == nullptr) {
129         return 0;
130     }
131     v->push();
132     if (is_right(v)) {
133         return count_left(v->l, is_right);
134     }
135     return (v->l != nullptr ? v->l->sz : 0) + 1 + count_left(v->r, is_right)
136         ;
137 }
138
139 node* add(node* r, node* v, const function<bool(node*)> &go_left) {
140     pair<node*, node*> p = split(r, go_left);
141     return merge(p.first, merge(v, p.second));
142 }
143
144 node* remove(node* v) { // returns the new root
145     v->push();
146     node* x = v->l;
147     node* y = v->r;
148     node* p = v->p;
149     v->l = v->r = v->p = nullptr;
150     v->push();
151     v->pull(); // now v might be reusable...
152     node* z = merge(x, y);
153     if (p == nullptr) {
154         if (z != nullptr) {
155             z->p = nullptr;
156         }
157     }

```

```

156     return z;
157 }
158 if (p->l == v) {
159     p->l = z;
160 }
161 if (p->r == v) {
162     p->r = z;
163 }
164 while (true) {
165     p->push();
166     p->pull();
167     if (p->p == nullptr) {
168         break;
169     }
170     p = p->p;
171 }
172 return p;
173 }
174
175 node* next(node* v) {
176     if (v->r == nullptr) {
177         while (v->p != nullptr && v->p->r == v) {
178             v = v->p;
179         }
180         return v->p;
181     }
182     v->push();
183     v = v->r;
184     while (v->l != nullptr) {
185         v->push();
186         v = v->l;
187     }
188     return v;
189 }
190
191 node* prev(node* v) {
192     if (v->l == nullptr) {
193         while (v->p != nullptr && v->p->l == v) {
194             v = v->p;
195         }
196         return v->p;
197     }

```

```

198     v->push();
199     v = v->l;
200     while (v->r != nullptr) {
201         v->push();
202         v = v->r;
203     }
204     return v;
205 }
206
207 int get_size(node* v) {
208     return (v != nullptr ? v->sz : 0);
209 }
210
211 template<typename... T>
212 void apply(node* v, T... args) {
213     v->unsafe_apply(args...);
214 }
215
216 void reverse(node* v) {
217     v->unsafe_reverse();
218 }
219
220 } // namespace treap
221
222 using namespace treap;

```

2 flows

2.1 blossom.cpp

```

1 template <typename T>
2 vector<int> find_max_unweighted_matching(const undigraph<T>& g) {
3     vector<int> mate(g.n, -1);
4     vector<int> label(g.n);
5     vector<int> parent(g.n);
6     vector<int> orig(g.n);
7     queue<int> q;
8     vector<int> aux(g.n, -1);
9     int aux_time = -1;
10    auto lca = [&](int x, int y) {
11        aux_time++;

```

```

12    while (true) {
13        if (x != -1) {
14            if (aux[x] == aux_time) {
15                return x;
16            }
17            aux[x] = aux_time;
18            if (mate[x] == -1) {
19                x = -1;
20            } else {
21                x = orig[parent[mate[x]]];
22            }
23        }
24        swap(x, y);
25    }
26 };
27 auto blossom = [&](int v, int w, int a) {
28     while (orig[v] != a) {
29         parent[v] = w;
30         w = mate[v];
31         if (label[w] == 1) {
32             label[w] = 0;
33             q.push(w);
34         }
35         orig[v] = orig[w] = a;
36         v = parent[w];
37     }
38 };
39 auto augment = [&](int v) {
40     while (v != -1) {
41         int pv = parent[v];
42         int nv = mate[pv];
43         mate[v] = pv;
44         mate[pv] = v;
45         v = nv;
46     }
47 };
48 auto bfs = [&](int root) {
49     fill(label.begin(), label.end(), -1);
50     iota(orig.begin(), orig.end(), 0);
51     while (!q.empty()) {
52         q.pop();
53     }

```

```

54     q.push(root);
55     label[root] = 0;
56     while (!q.empty()) {
57         int v = q.front();
58         q.pop();
59         for (int id : g.g[v]) {
60             auto &e = g.edges[id];
61             int x = e.from ^ e.to ^ v;
62             if (label[x] == -1) {
63                 label[x] = 1;
64                 parent[x] = v;
65                 if (mate[x] == -1) {
66                     augment(x);
67                     return true;
68                 }
69                 label[mate[x]] = 0;
70                 q.push(mate[x]);
71                 continue;
72             }
73             if (label[x] == 0 && orig[v] != orig[x]) {
74                 int a = lca(orig[v], orig[x]);
75                 blossom(x, v, a);
76                 blossom(v, x, a);
77             }
78         }
79     }
80     return false;
81 };
82 auto greedy = [&]() {
83     vector<int> order(g.n);
84     iota(order.begin(), order.end(), 0);
85     shuffle(order.begin(), order.end(), mt19937(787788));
86     for (int i : order) {
87         if (mate[i] == -1) {
88             for (int id : g.g[i]) {
89                 auto &e = g.edges[id];
90                 int to = e.from ^ e.to ^ i;
91                 if (i != to && mate[to] == -1) {
92                     mate[i] = to;
93                     mate[to] = i;
94                     break;
95                 }

```

```

96             }
97         }
98     }
99 };
100 greedy();
101 for (int i = 0; i < g.n; i++) {
102     if (mate[i] == -1) {
103         bfs(i);
104     }
105 }
106 return mate;
107 }

```

2.2 dinic-edge-ids.cpp

```

1  template <typename T>
2  class flow_graph {
3  public:
4      static constexpr T eps = (T) 1e-9;
5
6      struct edge {
7          int from;
8          int to;
9          T c;
10         T f;
11     };
12
13     vector<vector<int>>> g;
14     vector<edge> edges;
15     int n;
16     int st, fin;
17     T flow;
18
19     vector<int> ptr;
20     vector<int> d;
21     vector<int> q;
22
23     flow_graph(int _n, int _st, int _fin) : n(_n), st(_st), fin(_fin) {
24         assert(0 <= st && st < n && 0 <= fin && fin < n && st != fin);
25         g.resize(n);
26         ptr.resize(n);

```



```

27     d.resize(n);
28     q.resize(n);
29     flow = 0;
30 }
31
32 void clear_flow() {
33     for (const edge &e : edges) {
34         e.f = 0;
35     }
36     flow = 0;
37 }
38
39 void add(int from, int to, T forward_cap, T backward_cap) {
40     assert(0 <= from && from < n && 0 <= to && to < n);
41     g[from].push_back((int) edges.size());
42     edges.push_back({from, to, forward_cap, 0});
43     g[to].push_back((int) edges.size());
44     edges.push_back({to, from, backward_cap, 0});
45 }
46
47 bool expath() {
48     fill(d.begin(), d.end(), -1);
49     q[0] = fin;
50     d[fin] = 0;
51     int beg = 0, end = 1;
52     while (beg < end) {
53         int i = q[beg++];
54         for (int id : g[i]) {
55             const edge &e = edges[id];
56             const edge &back = edges[id ^ 1];
57             if (back.c - back.f > eps && d[e.to] == -1) {
58                 d[e.to] = d[i] + 1;
59                 if (e.to == st) {
60                     return true;
61                 }
62                 q[end++] = e.to;
63             }
64         }
65     }
66     return false;
67 }
68

```

```

69 T dfs(int v, T w) {
70     if (v == fin) {
71         return w;
72     }
73     int &j = ptr[v];
74     while (j >= 0) {
75         int id = g[v][j];
76         const edge &e = edges[id];
77         if (e.c - e.f > eps && d[e.to] == d[v] - 1) {
78             T t = dfs(e.to, min(e.c - e.f, w));
79             if (t > eps) {
80                 edges[id].f += t;
81                 edges[id ^ 1].f -= t;
82                 return t;
83             }
84         }
85         j--;
86     }
87     return 0;
88 }
89
90 T max_flow() {
91     while (expath()) {
92         for (int i = 0; i < n; i++) {
93             ptr[i] = (int) g[i].size() - 1;
94         }
95         T big_add = 0;
96         while (true) {
97             T add = dfs(st, numeric_limits<T>::max());
98             if (add <= eps) {
99                 break;
100             }
101             big_add += add;
102         }
103         if (big_add <= eps) {
104             break;
105         }
106         flow += big_add;
107     }
108     return flow;
109 }
110

```

```

111     vector<bool> min_cut() {
112         max_flow();
113         vector<bool> ret(n);
114         for (int i = 0; i < n; i++) {
115             ret[i] = (d[i] != -1);
116         }
117         return ret;
118     }
119
120     // Maximum flow / minimum cut, Dinic's algorithm
121     // Usage:
122     // 1) flow_graph<T> g(n, start, finish); [T == int / long long / double]
123     // 2) g.add(from, to, forward_cap, backward_cap);
124     // 3) cout << g.max_flow() << endl;
125     // 4) vector<bool> cut = g.min_cut();
126     //     for (auto &e : g.edges)
127     //         if (cut[e.from] != cut[e.to]) ; // edge e = (e.from -> e.to) is
128         // cut
129 };

```

2.3 dinic-old.cpp

```

1  template <typename T>
2  class flow_graph {
3  public:
4      static constexpr T eps = (T) 1e-9;
5
6      struct edge {
7          int to;
8          T c;
9          T f;
10         int rev;
11     };
12
13     vector<vector<edge>> g;
14     vector<int> ptr;
15     vector<int> d;
16     vector<int> q;
17     int n;
18     int st, fin;
19     T flow;

```

```

20
21     flow_graph(int _n, int _st, int _fin) : n(_n), st(_st), fin(_fin) {
22         assert(0 <= st && st < n && 0 <= fin && fin < n && st != fin);
23         g.resize(n);
24         ptr.resize(n);
25         d.resize(n);
26         q.resize(n);
27         flow = 0;
28     }
29
30     void clear_flow() {
31         for (int i = 0; i < n; i++) {
32             for (edge &e : g[i]) {
33                 e.f = 0;
34             }
35         }
36         flow = 0;
37     }
38
39     void add(int from, int to, T forward_cap, T backward_cap) {
40         assert(0 <= from && from < n && 0 <= to && to < n);
41         int from_size = g[from].size();
42         int to_size = g[to].size();
43         g[from].push_back({to, forward_cap, 0, to_size});
44         g[to].push_back({from, backward_cap, 0, from_size});
45     }
46
47     bool expath() {
48         fill(d.begin(), d.end(), -1);
49         q[0] = fin;
50         d[fin] = 0;
51         int beg = 0, end = 1;
52         while (beg < end) {
53             int i = q[beg++];
54             for (const edge &e : g[i]) {
55                 const edge &back = g[e.to][e.rev];
56                 if (back.c - back.f > eps && d[e.to] == -1) {
57                     d[e.to] = d[i] + 1;
58                     if (e.to == st) {
59                         return true;
60                     }
61                     q[end++] = e.to;

```

```

62     }
63     }
64     }
65     return false;
66 }
67
68 T dfs(int v, T w) {
69     if (v == fin) {
70         return w;
71     }
72     int &j = ptr[v];
73     while (j >= 0) {
74         const edge &e = g[v][j];
75         if (e.c - e.f > eps && d[e.to] == d[v] - 1) {
76             T t = dfs(e.to, min(e.c - e.f, w));
77             if (t > eps) {
78                 g[v][j].f += t;
79                 g[e.to][e.rev].f -= t;
80                 return t;
81             }
82         }
83         j--;
84     }
85     return 0;
86 }
87
88 T max_flow() {
89     while (expath()) {
90         for (int i = 0; i < n; i++) {
91             ptr[i] = (int) g[i].size() - 1;
92         }
93         T big_add = 0;
94         while (true) {
95             T add = dfs(st, numeric_limits<T>::max());
96             if (add <= eps) {
97                 break;
98             }
99             big_add += add;
100         }
101         if (big_add <= eps) {
102             break;
103         }

```

```

104         flow += big_add;
105     }
106     return flow;
107 }
108
109 vector<bool> min_cut() {
110     max_flow();
111     vector<bool> ret(n);
112     for (int i = 0; i < n; i++) {
113         ret[i] = (d[i] != -1);
114     }
115     return ret;
116 }
117 };

```

2.4 dinic.cpp

```

1 template <typename T>
2 class dinic {
3 public:
4     flow_graph<T> &g;
5
6     vector<int> ptr;
7     vector<int> d;
8     vector<int> q;
9
10    dinic(flow_graph<T> &_g) : g(_g) {
11        ptr.resize(g.n);
12        d.resize(g.n);
13        q.resize(g.n);
14    }
15
16    bool expath() {
17        fill(d.begin(), d.end(), -1);
18        q[0] = g.fin;
19        d[g.fin] = 0;
20        int beg = 0, end = 1;
21        while (beg < end) {
22            int i = q[beg++];
23            for (int id : g.g[i]) {
24                const auto &e = g.edges[id];

```

```

25     const auto &back = g.edges[id ^ 1];
26     if (back.c - back.f > g.eps && d[e.to] == -1) {
27         d[e.to] = d[i] + 1;
28         if (e.to == g.st) {
29             return true;
30         }
31         q[end++] = e.to;
32     }
33 }
34 }
35 return false;
36 }
37
38 T dfs(int v, T w) {
39     if (v == g.fin) {
40         return w;
41     }
42     int &j = ptr[v];
43     while (j >= 0) {
44         int id = g.g[v][j];
45         const auto &e = g.edges[id];
46         if (e.c - e.f > g.eps && d[e.to] == d[v] - 1) {
47             T t = dfs(e.to, min(e.c - e.f, w));
48             if (t > g.eps) {
49                 g.edges[id].f += t;
50                 g.edges[id ^ 1].f -= t;
51                 return t;
52             }
53         }
54         j--;
55     }
56     return 0;
57 }
58
59 T max_flow() {
60     while (expath()) {
61         for (int i = 0; i < g.n; i++) {
62             ptr[i] = (int) g.g[i].size() - 1;
63         }
64         T big_add = 0;
65         while (true) {
66             T add = dfs(g.st, numeric_limits<T>::max());

```

```

67             if (add <= g.eps) {
68                 break;
69             }
70             big_add += add;
71         }
72         if (big_add <= g.eps) {
73             break;
74         }
75         g.flow += big_add;
76     }
77     return g.flow;
78 }
79
80 vector<bool> min_cut() {
81     max_flow();
82     vector<bool> ret(g.n);
83     for (int i = 0; i < g.n; i++) {
84         ret[i] = (d[i] != -1);
85     }
86     return ret;
87 }
88 };

```

2.5 fastflow-other.cpp

```

1 // https://pastebin.com/exQM152L
2
3 // Doesn't walk through the whole path during augment at the cost of
4 // bigger constant
5 // Not recommended to use with double
6
7 template <typename T>
8 class flow_graph {
9     public:
10         static constexpr T eps = (T) 1e-9;
11
12         struct edge {
13             int to;
14             T c;
15             T f;
16             int rev;

```

```

16     };
17
18     vector<vector<edge>> g;
19     vector<int> ptr;
20     vector<int> d;
21     vector<int> q;
22     vector<int> cnt_on_layer;
23     vector<int> prev_edge;
24     vector<T> to_push;
25     vector<T> pushed;
26     vector<int> smallest;
27     bool can_reach_sink;
28
29     int n;
30     int st, fin;
31     T flow;
32
33     flow_graph(int _n, int _st, int _fin) : n(_n), st(_st), fin(_fin) {
34         assert(0 <= st && st < n && 0 <= fin && fin < n && st != fin);
35         g.resize(n);
36         ptr.resize(n);
37         d.resize(n);
38         q.resize(n);
39         cnt_on_layer.resize(n + 1);
40         prev_edge.resize(n);
41         to_push.resize(n);
42         pushed.resize(n);
43         smallest.resize(n);
44         flow = 0;
45     }
46
47     void clear_flow() {
48         for (int i = 0; i < n; i++) {
49             for (edge &e : g[i]) {
50                 e.f = 0;
51             }
52         }
53         flow = 0;
54     }
55
56     void add(int from, int to, T forward_cap, T backward_cap) {
57         assert(0 <= from && from < n && 0 <= to && to < n);

```

```

58         int from_size = g[from].size();
59         int to_size = g[to].size();
60         g[from].push_back({to, forward_cap, 0, to_size});
61         g[to].push_back({from, backward_cap, 0, from_size});
62     }
63
64     bool expath() {
65         fill(d.begin(), d.end(), n);
66         q[0] = fin;
67         d[fin] = 0;
68         fill(cnt_on_layer.begin(), cnt_on_layer.end(), 0);
69         cnt_on_layer[n] = n - 1;
70         cnt_on_layer[0] = 1;
71         int beg = 0, end = 1;
72         while (beg < end) {
73             int i = q[beg++];
74             for (const edge &e : g[i]) {
75                 const edge &back = g[e.to][e.rev];
76                 if (back.c - back.f > eps && d[e.to] == n) {
77                     cnt_on_layer[d[e.to]]--;
78                     d[e.to] = d[i] + 1;
79                     cnt_on_layer[d[e.to]]++;
80                     q[end++] = e.to;
81                 }
82             }
83         }
84         return (d[st] != n);
85     }
86
87     void rollback(int &v) {
88         edge &e = g[v][prev_edge[v]];
89         if (pushed[v]) {
90             edge &back = g[e.to][e.rev];
91             back.f += pushed[v];
92             e.f -= pushed[v];
93             pushed[e.to] += pushed[v];
94             to_push[e.to] -= pushed[v];
95             pushed[v] = 0;
96         }
97         v = e.to;
98     }
99

```

```

100 void augment(int &v) {
101     pushed[v] += to_push[v];
102     to_push[v] = 0;
103     int new_v = smallest[v];
104     while (v != new_v) {
105         rollback(v);
106     }
107 }
108
109 void retreat(int &v) {
110     int new_dist = n - 1;
111     for (const edge &e : g[v]) {
112         if (e.c - e.f > eps && d[e.to] < new_dist) {
113             new_dist = d[e.to];
114         }
115     }
116     cnt_on_layer[d[v]]--;
117     if (cnt_on_layer[d[v]] == 0) {
118         if (new_dist + 1 > d[v]) {
119             can_reach_sink = false;
120         }
121     }
122     d[v] = new_dist + 1;
123     cnt_on_layer[d[v]]++;
124     if (v != st) {
125         rollback(v);
126     }
127 }
128
129 T max_flow() {
130     can_reach_sink = true;
131     for (int i = 0; i < n; i++) {
132         ptr[i] = (int) g[i].size() - 1;
133     }
134     if (expath()) {
135         int v = st;
136         to_push[v] = numeric_limits<T>::max();
137         smallest[v] = v;
138         while (d[st] < n) {
139             while (ptr[v] >= 0) {
140                 const edge &e = g[v][ptr[v]];
141                 if (e.c - e.f > eps && d[e.to] == d[v] - 1) {

```

```

142                     prev_edge[e.to] = e.rev;
143                     to_push[e.to] = to_push[v];
144                     smallest[e.to] = smallest[v];
145                     if (e.c - e.f < to_push[e.to]) {
146                         to_push[e.to] = e.c - e.f;
147                         smallest[e.to] = v;
148                     }
149                     v = e.to;
150                     if (v == fin) {
151                         augment(v);
152                     }
153                     break;
154                 }
155                 ptr[v]--;
156             }
157             if (ptr[v] < 0) {
158                 ptr[v] = (int) g[v].size() - 1;
159                 retreat(v);
160                 if (!can_reach_sink) {
161                     break;
162                 }
163             }
164         }
165         while (v != st) {
166             rollback(v);
167         }
168         flow += pushed[st];
169         pushed[st] = 0;
170     }
171     return flow;
172 }
173
174 vector<bool> min_cut() {
175     max_flow();
176     assert(!expath());
177     vector<bool> ret(n);
178     for (int i = 0; i < n; i++) {
179         ret[i] = (d[i] != n);
180     }
181     return ret;
182 }
183 };

```

2.6 fastflow.cpp

```
1 // https://pastebin.com/exQM152L
2
3 template <typename T>
4 class flow_graph {
5 public:
6     static constexpr T eps = (T) 1e-9;
7
8     struct edge {
9         int to;
10        T c;
11        T f;
12        int rev;
13    };
14
15    vector<vector<edge>> g;
16    vector<int> ptr;
17    vector<int> d;
18    vector<int> q;
19    vector<int> cnt_on_layer;
20    vector<int> prev_edge;
21    bool can_reach_sink;
22
23    int n;
24    int st, fin;
25    T flow;
26
27    flow_graph(int _n, int _st, int _fin) : n(_n), st(_st), fin(_fin) {
28        assert(0 <= st && st < n && 0 <= fin && fin < n && st != fin);
29        g.resize(n);
30        ptr.resize(n);
31        d.resize(n);
32        q.resize(n);
33        cnt_on_layer.resize(n + 1);
34        prev_edge.resize(n);
35        flow = 0;
36    }
37
38    void clear_flow() {
39        for (int i = 0; i < n; i++) {
40            for (edge &e : g[i]) {
```

```
41                e.f = 0;
42            }
43        }
44        flow = 0;
45    }
46
47    void add(int from, int to, T forward_cap, T backward_cap) {
48        assert(0 <= from && from < n && 0 <= to && to < n);
49        int from_size = g[from].size();
50        int to_size = g[to].size();
51        g[from].push_back({to, forward_cap, 0, to_size});
52        g[to].push_back({from, backward_cap, 0, from_size});
53    }
54
55    bool expath() {
56        fill(d.begin(), d.end(), n);
57        q[0] = fin;
58        d[fin] = 0;
59        fill(cnt_on_layer.begin(), cnt_on_layer.end(), 0);
60        cnt_on_layer[n] = n - 1;
61        cnt_on_layer[0] = 1;
62        int beg = 0, end = 1;
63        while (beg < end) {
64            int i = q[beg++];
65            for (const edge &e : g[i]) {
66                const edge &back = g[e.to][e.rev];
67                if (back.c - back.f > eps && d[e.to] == n) {
68                    cnt_on_layer[d[e.to]]--;
69                    d[e.to] = d[i] + 1;
70                    cnt_on_layer[d[e.to]]++;
71                    q[end++] = e.to;
72                }
73            }
74        }
75        return (d[st] != n);
76    }
77
78    T augment(int &v) {
79        T cur = numeric_limits<T>::max();
80        int i = fin;
81        while (i != st) {
82            const edge &e = g[i][prev_edge[i]];
```

```

83     const edge &back = g[e.to][e.rev];
84     cur = min(cur, back.c - back.f);
85     i = e.to;
86 }
87 i = fin;
88 while (i != st) {
89     edge &e = g[i][prev_edge[i]];
90     edge &back = g[e.to][e.rev];
91     back.f += cur;
92     e.f -= cur;
93     i = e.to;
94     if (back.c - back.f <= eps) {
95         v = i;
96     }
97 }
98 return cur;
99 }
100
101 int retreat(int v) {
102     int new_dist = n - 1;
103     for (const edge &e : g[v]) {
104         if (e.c - e.f > eps && d[e.to] < new_dist) {
105             new_dist = d[e.to];
106         }
107     }
108     cnt_on_layer[d[v]]--;
109     if (cnt_on_layer[d[v]] == 0) {
110         if (new_dist + 1 > d[v]) {
111             can_reach_sink = false;
112         }
113     }
114     d[v] = new_dist + 1;
115     cnt_on_layer[d[v]]++;
116     if (v != st) {
117         v = g[v][prev_edge[v]].to;
118     }
119     return v;
120 }
121
122 T max_flow() {
123     can_reach_sink = true;
124     for (int i = 0; i < n; i++) {

```

```

125         ptr[i] = (int) g[i].size() - 1;
126     }
127     if (expath()) {
128         int v = st;
129         while (d[st] < n) {
130             while (ptr[v] >= 0) {
131                 const edge &e = g[v][ptr[v]];
132                 if (e.c - e.f > eps && d[e.to] == d[v] - 1) {
133                     prev_edge[e.to] = e.rev;
134                     v = e.to;
135                     if (v == fin) {
136                         flow += augment(v);
137                     }
138                     break;
139                 }
140                 ptr[v]--;
141             }
142             if (ptr[v] < 0) {
143                 ptr[v] = (int) g[v].size() - 1;
144                 v = retreat(v);
145                 if (!can_reach_sink) {
146                     break;
147                 }
148             }
149         }
150     }
151     return flow;
152 }
153
154 vector<bool> min_cut() {
155     max_flow();
156     assert(!expath());
157     vector<bool> ret(n);
158     for (int i = 0; i < n; i++) {
159         ret[i] = (d[i] != n);
160     }
161     return ret;
162 }
163 };

```


2.7 flow-decomposition.cpp

```
1  template <typename T>
2  class flow_decomposition {
3  public:
4      const flow_graph<T> &g;
5
6      vector<vector<int>>> paths;
7      vector<T> path_flows;
8      vector<vector<int>>> cycles;
9      vector<T> cycle_flows;
10
11     flow_decomposition(const flow_graph<T> &g) : g(_g) {
12     }
13
14     void decompose() {
15         vector<T> fs(g.edges.size());
16         for (int i = 0; i < (int) g.edges.size(); i++) {
17             fs[i] = g.edges[i].f;
18         }
19         paths.clear();
20         path_flows.clear();
21         cycles.clear();
22         cycle_flows.clear();
23         vector<int> ptr(g.n);
24         for (int i = 0; i < g.n; i++) {
25             ptr[i] = (int) g.g[i].size() - 1;
26         }
27         vector<int> was(g.n, -1);
28         int start = g.st;
29         for (int iter = 0; ; iter++) {
30             bool found_start = false;
31             while (true) {
32                 if (ptr[start] >= 0) {
33                     int id = g.g[start][ptr[start]];
34                     if (fs[id] > g.eps) {
35                         found_start = true;
36                         break;
37                     }
38                     ptr[start]--;
39                     continue;
40                 }
```

```
41                 start = (start + 1) % g.n;
42                 if (start == g.st) {
43                     break;
44                 }
45             }
46             if (!found_start) {
47                 break;
48             }
49             vector<int> path;
50             bool is_cycle = false;
51             int v = start;
52             while (true) {
53                 if (v == g.fin) {
54                     break;
55                 }
56                 if (was[v] == iter) {
57                     bool found = false;
58                     for (int i = 0; i < (int) path.size(); i++) {
59                         int id = path[i];
60                         auto &e = g.edges[id];
61                         if (e.from == v) {
62                             path.erase(path.begin(), path.begin() + i);
63                             found = true;
64                             break;
65                         }
66                     }
67                     assert(found);
68                     is_cycle = true;
69                     break;
70                 }
71                 was[v] = iter;
72                 bool found = false;
73                 while (ptr[v] >= 0) {
74                     int id = g.g[v][ptr[v]];
75                     if (fs[id] > g.eps) {
76                         path.push_back(id);
77                         v = g.edges[id].to;
78                         found = true;
79                         break;
80                     }
81                     ptr[v]--;
82                 }
```

```

83     assert(found);
84 }
85 T path_flow = numeric_limits<T>::max();
86 for (int id : path) {
87     path_flow = min(path_flow, fs[id]);
88 }
89 for (int id : path) {
90     fs[id] -= path_flow;
91     fs[id ^ 1] += path_flow;
92 }
93 if (is_cycle) {
94     cycles.push_back(path);
95     cycle_flows.push_back(path_flow);
96 } else {
97     paths.push_back(path);
98     path_flows.push_back(path_flow);
99 }
100 }
101 for (const T& f : fs) {
102     assert(-g.eps <= f && f <= g.eps);
103 }
104 }
105 };

```

2.8 flow-graph.cpp

```

1  template <typename T>
2  class flow_graph {
3  public:
4      static constexpr T eps = (T) 1e-9;
5
6      struct edge {
7          int from;
8          int to;
9          T c;
10         T f;
11     };
12
13     vector<vector<int>>> g;
14     vector<edge> edges;
15     int n;

```

```

16     int st;
17     int fin;
18     T flow;
19
20     flow_graph(int _n, int _st, int _fin) : n(_n), st(_st), fin(_fin) {
21         assert(0 <= st && st < n && 0 <= fin && fin < n && st != fin);
22         g.resize(n);
23         flow = 0;
24     }
25
26     void clear_flow() {
27         for (const edge &e : edges) {
28             e.f = 0;
29         }
30         flow = 0;
31     }
32
33     int add(int from, int to, T forward_cap, T backward_cap) {
34         assert(0 <= from && from < n && 0 <= to && to < n);
35         int id = (int) edges.size();
36         g[from].push_back(id);
37         edges.push_back({from, to, forward_cap, 0});
38         g[to].push_back(id + 1);
39         edges.push_back({to, from, backward_cap, 0});
40         return id;
41     }
42 };

```

2.9 gomory-hu-old.cpp

```

1  template <typename T>
2  forest<T> gomory_hu(const undigraph<T> &g) {
3      int n = g.n;
4      if (n == 1) {
5          return forest<T>(n);
6      }
7      flow_graph<T> fg(n, 0, 1);
8      for (auto &e : g.edges) {
9          fg.add(e.from, e.to, e.cost, e.cost);
10     }
11     vector<vector<int>>> dist(n, vector<int>(n, numeric_limits<T>::max()));

```

```

12 function<void(vector<int>>> dfs = [&g, &n, &fg, &dist, &dfs](vector<int>
    group) {
13     int sz = group.size();
14     if (sz == 1) {
15         return;
16     }
17     fg.clear_flow();
18     fg.st = group[0];
19     fg.fin = group[1];
20     T flow = fg.max_flow();
21     vector<bool> cut = fg.min_cut();
22     for (int i = 0; i < n; i++) {
23         for (int j = i + 1; j < n; j++) {
24             if (cut[i] != cut[j]) {
25                 dist[i][j] = min(dist[i][j], flow);
26             }
27         }
28     }
29     vector<int> new_groups[2];
30     for (int v : group) {
31         new_groups[(int) cut[v]].push_back(v);
32     }
33     for (int id = 0; id < 2; id++) {
34         dfs(new_groups[id]);
35     }
36 };
37 vector<int> group(n);
38 iota(group.begin(), group.end(), 0);
39 dfs(group);
40 undigraph<T> mg(n);
41 for (int i = 0; i < n; i++) {
42     for (int j = i + 1; j < n; j++) {
43         mg.add(i, j, -dist[i][j]);
44     }
45 }
46 T foo;
47 vector<int> ids = mst(mg, foo);
48 forest<T> ret(n);
49 for (int id : ids) {
50     auto &e = mg.edges[id];
51     ret.add(e.from, e.to, -e.cost);
52 }

```

```

53     return ret;
54     // don't be lazy next time!
55     // implement a proper gomory-hu tree
56 }

```

2.10 gomory-hu.cpp

```

1 template <typename T>
2 forest<T> gomory_hu(const undigraph<T>& g) {
3     int n = g.n;
4     flow_graph<T> fg(n, 0, 1);
5     for (auto& e : g.edges) {
6         fg.add(e.from, e.to, e.cost, e.cost);
7     }
8     forest<T> ret(n);
9     vector<int> pr(n, 0);
10    for (int i = 1; i < n; i++) {
11        fg.clear_flow();
12        fg.st = i;
13        fg.fin = pr[i];
14        T flow = fg.max_flow();
15        vector<bool> cut = fg.min_cut();
16        for (int j = i + 1; j < n; j++) {
17            if (cut[j] == cut[i] && pr[j] == pr[i]) {
18                pr[j] = i;
19            }
20        }
21        ret.add(i, pr[i], flow);
22    }
23    return ret;
24    // can be optimized by compressing components
25 }

```

2.11 hungarian-arrays.cpp

```

1 template <typename T>
2 class hungarian {
3 public:
4     static const int MAX_N = ... + 1;
5
6     int n;

```

```

7   int m;
8   T a[MAX_N][MAX_N];
9   T u[MAX_N];
10  T v[MAX_N];
11  int pa[MAX_N];
12  int pb[MAX_N];
13  int way[MAX_N];
14  T minv[MAX_N];
15  bool used[MAX_N];
16  T inf;
17
18  hungarian(int _n, int _m) : n(_n), m(_m) {
19      assert(n <= m);
20      T zero = T{};
21      fill(u, u + n + 1, zero);
22      fill(v, v + m + 1, zero);
23      fill(pa, pa + n + 1, -1);
24      fill(pb, pb + m + 1, -1);
25      inf = numeric_limits<T>::max();
26  }
27
28  inline void add_row(int i) {
29      fill(minv, minv + m + 1, inf);
30      fill(used, used + m + 1, false);
31      pb[m] = i;
32      pa[i] = m;
33      int j0 = m;
34      do {
35          used[j0] = true;
36          int i0 = pb[j0];
37          T delta = inf;
38          int j1 = -1;
39          for (int j = 0; j < m; j++) {
40              if (!used[j]) {
41                  T cur = a[i0][j] - u[i0] - v[j];
42                  if (cur < minv[j]) {
43                      minv[j] = cur;
44                      way[j] = j0;
45                  }
46                  if (minv[j] < delta) {
47                      delta = minv[j];
48                      j1 = j;

```

```

49              }
50          }
51      }
52      for (int j = 0; j <= m; j++) {
53          if (used[j]) {
54              u[pb[j]] += delta;
55              v[j] -= delta;
56          } else {
57              minv[j] -= delta;
58          }
59      }
60      j0 = j1;
61  } while (pb[j0] != -1);
62  do {
63      int j1 = way[j0];
64      pb[j0] = pb[j1];
65      pa[pb[j0]] = j0;
66      j0 = j1;
67  } while (j0 != m);
68  }
69
70  inline T current_score() {
71      return -v[m];
72  }
73
74  inline T solve() {
75      for (int i = 0; i < n; i++) {
76          add_row(i);
77      }
78      return current_score();
79  }
80  };

```

2.12 hungarian.cpp

```

1  template <typename T>
2  class hungarian {
3  public:
4      int n;
5      int m;
6      vector<vector<T>>> a;

```

```

7   vector<T> u;
8   vector<T> v;
9   vector<int> pa;
10  vector<int> pb;
11  vector<int> way;
12  vector<T> minv;
13  vector<bool> used;
14  T inf;
15
16  hungarian(int _n, int _m) : n(_n), m(_m) {
17      assert(n <= m);
18      a = vector<vector<T>>(n, vector<T>(m));
19      u = vector<T>(n + 1);
20      v = vector<T>(m + 1);
21      pa = vector<int>(n + 1, -1);
22      pb = vector<int>(m + 1, -1);
23      way = vector<int>(m, -1);
24      minv = vector<T>(m);
25      used = vector<bool>(m + 1);
26      inf = numeric_limits<T>::max();
27  }
28
29  inline void add_row(int i) {
30      fill(minv.begin(), minv.end(), inf);
31      fill(used.begin(), used.end(), false);
32      pb[m] = i;
33      pa[i] = m;
34      int j0 = m;
35      do {
36          used[j0] = true;
37          int i0 = pb[j0];
38          T delta = inf;
39          int j1 = -1;
40          for (int j = 0; j < m; j++) {
41              if (!used[j]) {
42                  T cur = a[i0][j] - u[i0] - v[j];
43                  if (cur < minv[j]) {
44                      minv[j] = cur;
45                      way[j] = j0;
46                  }
47                  if (minv[j] < delta) {
48                      delta = minv[j];

```

```

49                      j1 = j;
50                  }
51              }
52          }
53          for (int j = 0; j <= m; j++) {
54              if (used[j]) {
55                  u[pb[j]] += delta;
56                  v[j] -= delta;
57              } else {
58                  minv[j] -= delta;
59              }
60          }
61          j0 = j1;
62      } while (pb[j0] != -1);
63      do {
64          int j1 = way[j0];
65          pb[j0] = pb[j1];
66          pa[pb[j0]] = j0;
67          j0 = j1;
68      } while (j0 != m);
69  }
70
71  inline T current_score() {
72      return -v[m];
73  }
74
75  inline T solve() {
76      for (int i = 0; i < n; i++) {
77          add_row(i);
78      }
79      return current_score();
80  }
81  };

```

2.13 matching.cpp

```

1  class matching {
2  public:
3      vector<vector<int>> g;
4      vector<int> pa;
5      vector<int> pb;

```

```

6   vector<int> was;
7   int n, m;
8   int res;
9   int iter;
10
11  matching(int _n, int _m) : n(_n), m(_m) {
12      assert(0 <= n && 0 <= m);
13      pa = vector<int>(n, -1);
14      pb = vector<int>(m, -1);
15      was = vector<int>(n, 0);
16      g.resize(n);
17      res = 0;
18      iter = 0;
19  }
20
21  void add(int from, int to) {
22      assert(0 <= from && from < n && 0 <= to && to < m);
23      g[from].push_back(to);
24  }
25
26  bool dfs(int v) {
27      was[v] = iter;
28      for (int u : g[v]) {
29          if (pb[u] == -1) {
30              pa[v] = u;
31              pb[u] = v;
32              return true;
33          }
34      }
35      for (int u : g[v]) {
36          if (was[pb[u]] != iter && dfs(pb[u])) {
37              pa[v] = u;
38              pb[u] = v;
39              return true;
40          }
41      }
42      return false;
43  }
44
45  int solve() {
46      while (true) {
47          iter++;

```

```

48          int add = 0;
49          for (int i = 0; i < n; i++) {
50              if (pa[i] == -1 && dfs(i)) {
51                  add++;
52              }
53          }
54          if (add == 0) {
55              break;
56          }
57          res += add;
58      }
59      return res;
60  }
61
62  int run_one(int v) {
63      if (pa[v] != -1) {
64          return 0;
65      }
66      iter++;
67      return (int) dfs(v);
68  }
69  };

```

2.14 mcmf-slow.cpp

```

1  template <typename T, typename C>
2  class mcmf {
3  public:
4      static constexpr T eps = (T) 1e-9;
5
6      struct edge {
7          int from;
8          int to;
9          T c;
10         T f;
11         C cost;
12     };
13
14     vector<vector<int>> g;
15     vector<edge> edges;
16     vector<C> d;

```

```

17  vector<int> q;
18  vector<bool> in_queue;
19  vector<int> pe;
20  int n;
21  int st, fin;
22  T flow;
23  C cost;
24
25  mcmf(int _n, int _st, int _fin) : n(_n), st(_st), fin(_fin) {
26      assert(0 <= st && st < n && 0 <= fin && fin < n && st != fin);
27      g.resize(n);
28      d.resize(n);
29      in_queue.resize(n);
30      pe.resize(n);
31      flow = 0;
32      cost = 0;
33  }
34
35  void clear_flow() {
36      for (const edge &e : edges) {
37          e.f = 0;
38      }
39      flow = 0;
40  }
41
42  void add(int from, int to, T forward_cap, T backward_cap, C cost) {
43      assert(0 <= from && from < n && 0 <= to && to < n);
44      g[from].push_back((int) edges.size());
45      edges.push_back({from, to, forward_cap, 0, cost});
46      g[to].push_back((int) edges.size());
47      edges.push_back({to, from, backward_cap, 0, -cost});
48  }
49
50  bool expath() {
51      fill(d.begin(), d.end(), numeric_limits<C>::max());
52      q.clear();
53      q.push_back(st);
54      d[st] = 0;
55      in_queue[st] = true;
56      int beg = 0;
57      bool found = false;
58      while (beg < (int) q.size()) {

```

```

59          int i = q[beg++];
60          if (i == fin) {
61              found = true;
62          }
63          in_queue[i] = false;
64          for (int id : g[i]) {
65              const edge &e = edges[id];
66              if (e.c - e.f > eps && d[i] + e.cost < d[e.to]) {
67                  d[e.to] = d[i] + e.cost;
68                  pe[e.to] = id;
69                  if (!in_queue[e.to]) {
70                      q.push_back(e.to);
71                      in_queue[e.to] = true;
72                  }
73              }
74          }
75      }
76      if (found) {
77          T push = numeric_limits<T>::max();
78          int v = fin;
79          while (v != st) {
80              const edge &e = edges[pe[v]];
81              push = min(push, e.c - e.f);
82              v = e.from;
83          }
84          v = fin;
85          while (v != st) {
86              edge &e = edges[pe[v]];
87              e.f += push;
88              edge &back = edges[pe[v] ^ 1];
89              back.f -= push;
90              v = e.from;
91          }
92          flow += push;
93          cost += push * d[fin];
94      }
95      return found;
96  }
97
98  pair<T, C> max_flow_min_cost() {
99      while (expath()) {}
100     return {flow, cost};

```

```

101     }
102 };

```

2.15 mcmf.cpp

```

1  #include <bits/extc++.h>
2
3  template <typename T, typename C>
4  class MCMF {
5  public:
6      static constexpr T eps = (T) 1e-9;
7
8      struct edge {
9          int from;
10         int to;
11         T c;
12         T f;
13         C cost;
14     };
15
16     int n;
17     vector<vector<int>>> g;
18     vector<edge> edges;
19     vector<C> d;
20     vector<C> pot;
21     __gnu_pbds::priority_queue<pair<C, int>> q;
22     vector<typename decltype(q)::point_iterator> its;
23     vector<int> pe;
24     const C INF_C = numeric_limits<C>::max() / 2;
25
26     explicit MCMF(int n_) : n(n_), g(n), d(n), pot(n, 0), its(n), pe(n) {}
27
28     int add(int from, int to, T forward_cap, T backward_cap, C edge_cost) {
29         assert(0 <= from && from < n && 0 <= to && to < n);
30         assert(forward_cap >= 0 && backward_cap >= 0);
31         int id = static_cast<int>(edges.size());
32         g[from].push_back(id);
33         edges.push_back({from, to, forward_cap, 0, edge_cost});
34         g[to].push_back(id + 1);
35         edges.push_back({to, from, backward_cap, 0, -edge_cost});
36         return id;

```

```

37     }
38
39     void expath(int st) {
40         fill(d.begin(), d.end(), INF_C);
41         q.clear();
42         fill(its.begin(), its.end(), q.end());
43         its[st] = q.push({pot[st], st});
44         d[st] = 0;
45         while (!q.empty()) {
46             int i = q.top().second;
47             q.pop();
48             its[i] = q.end();
49             for (int id : g[i]) {
50                 const edge &e = edges[id];
51                 int j = e.to;
52                 if (e.c - e.f > eps && d[i] + e.cost < d[j]) {
53                     d[j] = d[i] + e.cost;
54                     pe[j] = id;
55                     if (its[j] == q.end()) {
56                         its[j] = q.push({pot[j] - d[j], j});
57                     } else {
58                         q.modify(its[j], {pot[j] - d[j], j});
59                     }
60                 }
61             }
62         }
63         swap(d, pot);
64     }
65
66     pair<T, C> max_flow_min_cost(int st, int fin) {
67         T flow = 0;
68         C cost = 0;
69         bool ok = true;
70         for (auto& e : edges) {
71             if (e.c - e.f > eps && e.cost + pot[e.from] - pot[e.to] < 0) {
72                 ok = false;
73                 break;
74             }
75         }
76         if (ok) {
77             expath(st);
78         } else {

```



```

79     vector<int> deg(n, 0);
80     for (int i = 0; i < n; i++) {
81         for (int eid : g[i]) {
82             auto& e = edges[eid];
83             if (e.c - e.f > eps) {
84                 deg[e.to] += 1;
85             }
86         }
87     }
88     vector<int> que;
89     for (int i = 0; i < n; i++) {
90         if (deg[i] == 0) {
91             que.push_back(i);
92         }
93     }
94     for (int b = 0; b < (int) que.size(); b++) {
95         for (int eid : g[que[b]]) {
96             auto& e = edges[eid];
97             if (e.c - e.f > eps) {
98                 deg[e.to] -= 1;
99                 if (deg[e.to] == 0) {
100                     que.push_back(e.to);
101                 }
102             }
103         }
104     }
105     fill(pot.begin(), pot.end(), INF_C);
106     pot[st] = 0;
107     if (static_cast<int>(que.size()) == n) {
108         for (int v : que) {
109             if (pot[v] < INF_C) {
110                 for (int eid : g[v]) {
111                     auto& e = edges[eid];
112                     if (e.c - e.f > eps) {
113                         if (pot[v] + e.cost < pot[e.to]) {
114                             pot[e.to] = pot[v] + e.cost;
115                             pe[e.to] = eid;
116                         }
117                     }
118                 }
119             }
120         }

```

```

121     } else {
122         que.assign(1, st);
123         vector<bool> in_queue(n, false);
124         in_queue[st] = true;
125         for (int b = 0; b < (int) que.size(); b++) {
126             int i = que[b];
127             in_queue[i] = false;
128             for (int id : g[i]) {
129                 const edge &e = edges[id];
130                 if (e.c - e.f > eps && pot[i] + e.cost < pot[e.to]) {
131                     pot[e.to] = pot[i] + e.cost;
132                     pe[e.to] = id;
133                     if (!in_queue[e.to]) {
134                         que.push_back(e.to);
135                         in_queue[e.to] = true;
136                     }
137                 }
138             }
139         }
140     }
141 }
142 while (pot[fin] < INF_C) {
143     T push = numeric_limits<T>::max();
144     int v = fin;
145     while (v != st) {
146         const edge &e = edges[pe[v]];
147         push = min(push, e.c - e.f);
148         v = e.from;
149     }
150     v = fin;
151     while (v != st) {
152         edge &e = edges[pe[v]];
153         e.f += push;
154         edge &back = edges[pe[v] ^ 1];
155         back.f -= push;
156         v = e.from;
157     }
158     flow += push;
159     cost += push * pot[fin];
160     expath(st);
161 }
162 return {flow, cost};

```

```

163     }
164 };

```

2.16 mincut.cpp

```

1  template <typename T>
2  pair<T, vector<bool>> MinCut(vector<vector<T>> g) {
3      int n = static_cast<int>(g.size());
4      for (int i = 0; i < n; i++) {
5          assert(static_cast<int>(g[i].size()) == n);
6      }
7      for (int i = 0; i < n; i++) {
8          for (int j = i + 1; j < n; j++) {
9              assert(g[i][j] == g[j][i]);
10         }
11     }
12     vector<vector<bool>> v(n, vector<bool>(n));
13     for (int i = 0; i < n; i++) {
14         v[i][i] = true;
15     }
16     vector<T> w(n);
17     vector<bool> exists(n, true);
18     vector<bool> in_a(n);
19     T best_cost = numeric_limits<T>::max();
20     vector<bool> best_cut;
21     for (int ph = 0; ph < n - 1; ph++) {
22         fill(in_a.begin(), in_a.end(), false);
23         fill(w.begin(), w.end(), T(0));
24         int prev = -1;
25         for (int it = 0; it < n - ph; it++) {
26             int sel = -1;
27             for (int i = 0; i < n; i++) {
28                 if (exists[i] && !in_a[i] && (sel == -1 || w[i] > w[sel])) {
29                     sel = i;
30                 }
31             }
32             if (it == n - ph - 1) {
33                 if (w[sel] < best_cost) {
34                     best_cost = w[sel];
35                     best_cut = v[sel];
36                 }

```

```

37         for (int i = 0; i < n; i++) {
38             v[prev][i] = v[prev][i] | v[sel][i];
39             g[prev][i] += g[sel][i];
40             g[i][prev] += g[i][sel];
41         }
42         exists[sel] = false;
43         break;
44     }
45     in_a[sel] = true;
46     for (int i = 0; i < n; i++) {
47         w[i] += g[sel][i];
48     }
49     prev = sel;
50 }
51 }
52 return make_pair(best_cost, best_cut);
53 }

```

3 graph

3.1 bicone.cpp

```

1  template <typename T>
2  vector<int> find_bicone(dfs_undigraph<T> &g, int &cnt) {
3      g.dfs_all();
4      vector<int> vertex_comp(g.n);
5      cnt = 0;
6      for (int i : g.order) {
7          if (g.pv[i] == -1 || g.min_depth[i] == g.depth[i]) {
8              vertex_comp[i] = cnt++;
9          } else {
10             vertex_comp[i] = vertex_comp[g.pv[i]];
11         }
12     }
13     return vertex_comp;
14 }

```

3.2 biconv.cpp

```

1  template <typename T>
2  vector<int> find_biconv(dfs_undigraph<T> &g, int &cnt) {

```

```

3   g.dfs_all();
4   vector<int> vertex_comp(g.n);
5   cnt = 0;
6   for (int i : g.order) {
7       if (g.pv[i] == -1) {
8           vertex_comp[i] = -1;
9           continue;
10      }
11      if (g.min_depth[i] >= g.depth[g.pv[i]]) {
12          vertex_comp[i] = cnt++;
13      } else {
14          vertex_comp[i] = vertex_comp[g.pv[i]];
15      }
16  }
17  vector<int> edge_comp(g.edges.size(), -1);
18  for (int id = 0; id < (int) g.edges.size(); id++) {
19      int x = g.edges[id].from;
20      int y = g.edges[id].to;
21      int z = (g.depth[x] > g.depth[y] ? x : y);
22      edge_comp[id] = vertex_comp[z];
23  }
24  return edge_comp;
25 }

```

3.3 bridges.cpp

```

1  template <typename T>
2  vector<bool> find_bridges(dfs_undigraph<T> &g) {
3      g.dfs_all();
4      vector<bool> bridge(g.edges.size(), false);
5      for (int i = 0; i < g.n; i++) {
6          if (g.pv[i] != -1 && g.min_depth[i] == g.depth[i]) {
7              bridge[g.pe[i]] = true;
8          }
9      }
10     return bridge;
11 }

```

3.4 cutpoints.cpp

```

1  template <typename T>

```

```

2  vector<bool> find_cutpoints(dfs_undigraph<T> &g) {
3      g.dfs_all();
4      vector<bool> cutpoint(g.n, false);
5      for (int i = 0; i < g.n; i++) {
6          if (g.pv[i] != -1 && g.min_depth[i] >= g.depth[g.pv[i]]) {
7              cutpoint[g.pv[i]] = true;
8          }
9      }
10     vector<int> children(g.n, 0);
11     for (int i = 0; i < g.n; i++) {
12         if (g.pv[i] != -1) {
13             children[g.pv[i]]++;
14         }
15     }
16     for (int i = 0; i < g.n; i++) {
17         if (g.pv[i] == -1 && children[i] < 2) {
18             cutpoint[i] = false;
19         }
20     }
21     return cutpoint;
22 }

```

3.5 cycles.cpp

```

1  template <typename T>
2  vector<vector<int>>> find_cycles(const graph<T> &g, int bound_cnt = 1 <<
3      30, int bound_size = 1 << 30) {
4      vector<int> was(g.n, -1);
5      vector<int> st;
6      vector<vector<int>>> cycles;
7      int total_size = 0;
8      function<void(int, int)> dfs = [&](int v, int pe) {
9          if ((int) cycles.size() >= bound_cnt || total_size >= bound_size) {
10              return;
11          }
12          was[v] = (int) st.size();
13          for (int id : g.g[v]) {
14              if (id == pe) {
15                  continue;
16              }
17              auto &e = g.edges[id];

```

```

17     int to = e.from ^ e.to ^ v;
18     if (was[to] >= 0) {
19         vector<int> cycle(1, id);
20         for (int j = was[to]; j < (int) st.size(); j++) {
21             cycle.push_back(st[j]);
22         }
23         cycles.push_back(cycle);
24         total_size += (int) cycle.size();
25         if ((int) cycles.size() >= bound_cnt || total_size >= bound_size)
26             {
27                 was[v] = -2;
28                 return;
29             }
30         continue;
31     }
32     if (was[to] == -1) {
33         st.push_back(id);
34         dfs(to, id);
35         st.pop_back();
36     }
37     was[v] = -2;
38 };
39 for (int i = 0; i < g.n; i++) {
40     if (was[i] == -1) {
41         dfs(i, -1);
42     }
43 }
44 return cycles;
45 // cycles are given by edge ids, all cycles are simple
46 // breaks after getting bound_cnt cycles or total_size >= bound_size
47 // digraph: finds at least one cycle in every connected component (if
48 not broken)
49 // undigraph: finds cycle basis
50 }
51 template <typename T>
52 vector<int> edges_to_vertices(const graph<T> &g, const vector<int> &
53     edge_cycle) {
54     int sz = (int) edge_cycle.size();
55     vector<int> vertex_cycle;
56     if (sz <= 2) {

```

```

56         vertex_cycle.push_back(g.edges[edge_cycle[0]].from);
57         if (sz == 2) {
58             vertex_cycle.push_back(g.edges[edge_cycle[0]].to);
59         }
60     } else {
61         for (int i = 0; i < sz; i++) {
62             int j = (i + 1) % sz;
63             auto &e = g.edges[edge_cycle[i]];
64             auto &other = g.edges[edge_cycle[j]];
65             if (other.from == e.from || other.to == e.from) {
66                 vertex_cycle.push_back(e.to);
67             } else {
68                 vertex_cycle.push_back(e.from);
69             }
70         }
71     }
72     return vertex_cycle;
73     // only for simple cycles!
74 }

```

3.6 dfs-digraph-useless.cpp

```

1 template <typename T>
2 class dfs_digraph : public digraph<T> {
3 public:
4     using digraph<T>::edges;
5     using digraph<T>::g;
6     using digraph<T>::n;
7
8     vector<int> pv;
9     vector<int> pe;
10    vector<int> order;
11    vector<int> pos;
12    vector<int> end;
13    vector<int> sz;
14    vector<int> root;
15    vector<int> depth;
16    vector<T> dist;
17
18    dfs_digraph(int _n) : digraph<T>(_n) {
19    }

```

```

20
21 void clear() {
22     pv.clear();
23     pe.clear();
24     order.clear();
25     pos.clear();
26     end.clear();
27     sz.clear();
28     root.clear();
29     depth.clear();
30     dist.clear();
31 }
32
33 void init() {
34     pv = vector<int>(n, -1);
35     pe = vector<int>(n, -1);
36     order.clear();
37     pos = vector<int>(n, -1);
38     end = vector<int>(n, -1);
39     sz = vector<int>(n, 0);
40     root = vector<int>(n, -1);
41     depth = vector<int>(n, -1);
42     dist = vector<T>(n);
43 }
44
45 private:
46 void do_dfs(int v) {
47     pos[v] = (int) order.size();
48     order.push_back(v);
49     sz[v] = 1;
50     for (int id : g[v]) {
51         if (id == pe[v]) {
52             continue;
53         }
54         auto &e = edges[id];
55         int to = e.from ^ e.to ^ v;
56         // well, this is controversial...
57         if (depth[to] != -1) {
58             continue;
59         }
60         depth[to] = depth[v] + 1;
61         dist[to] = dist[v] + e.cost;

```

```

62         pv[to] = v;
63         pe[to] = id;
64         root[to] = (root[v] != -1 ? root[v] : to);
65         do_dfs(to);
66         sz[v] += sz[to];
67     }
68     end[v] = (int) order.size() - 1;
69 }
70
71 void do_dfs_from(int v) {
72     depth[v] = 0;
73     dist[v] = T{};
74     root[v] = v;
75     pv[v] = pe[v] = -1;
76     do_dfs(v);
77 }
78
79 public:
80 int dfs_one_unsafe(int v) {
81     // run init() before this
82     // then run this with the required v's
83     do_dfs_from(v);
84     return v;
85 }
86
87 int dfs(int v) {
88     init();
89     do_dfs_from(v);
90     // assert((int) order.size() == n);
91     return v;
92 }
93
94 void dfs_many(const vector<int> &roots) {
95     init();
96     for (int v : roots) {
97         if (depth[v] == -1) {
98             do_dfs_from(v);
99         }
100     }
101     // assert((int) order.size() == n);
102 }
103

```

```

104     vector<int> dfs_all() {
105         init();
106         vector<int> roots;
107         for (int v = 0; v < n; v++) {
108             if (depth[v] == -1) {
109                 roots.push_back(v);
110                 do_dfs_from(v);
111             }
112         }
113         assert((int) order.size() == n);
114         return roots;
115     }
116 };

```

3.7 dfs-forest.cpp

```

1  template <typename T>
2  class dfs_forest : public forest<T> {
3  public:
4      using forest<T>::edges;
5      using forest<T>::g;
6      using forest<T>::n;
7
8      vector<int> pv;
9      vector<int> pe;
10     vector<int> order;
11     vector<int> pos;
12     vector<int> end;
13     vector<int> sz;
14     vector<int> root;
15     vector<int> depth;
16     vector<T> dist;
17
18     dfs_forest(int _n) : forest<T>(_n) {
19     }
20
21     void init() {
22         pv = vector<int>(n, -1);
23         pe = vector<int>(n, -1);
24         order.clear();
25         pos = vector<int>(n, -1);

```

```

26         end = vector<int>(n, -1);
27         sz = vector<int>(n, 0);
28         root = vector<int>(n, -1);
29         depth = vector<int>(n, -1);
30         dist = vector<T>(n);
31     }
32
33     void clear() {
34         pv.clear();
35         pe.clear();
36         order.clear();
37         pos.clear();
38         end.clear();
39         sz.clear();
40         root.clear();
41         depth.clear();
42         dist.clear();
43     }
44
45     private:
46     void do_dfs(int v) {
47         pos[v] = (int) order.size();
48         order.push_back(v);
49         sz[v] = 1;
50         for (int id : g[v]) {
51             if (id == pe[v]) {
52                 continue;
53             }
54             auto &e = edges[id];
55             int to = e.from ^ e.to ^ v;
56             depth[to] = depth[v] + 1;
57             dist[to] = dist[v] + e.cost;
58             pv[to] = v;
59             pe[to] = id;
60             root[to] = (root[v] != -1 ? root[v] : to);
61             do_dfs(to);
62             sz[v] += sz[to];
63         }
64         end[v] = (int) order.size() - 1;
65     }
66
67     void do_dfs_from(int v) {

```

```

68     depth[v] = 0;
69     dist[v] = T{};
70     root[v] = v;
71     pv[v] = pe[v] = -1;
72     do_dfs(v);
73 }
74
75 public:
76     void dfs(int v, bool clear_order = true) {
77         if (pv.empty()) {
78             init();
79         } else {
80             if (clear_order) {
81                 order.clear();
82             }
83         }
84         do_dfs_from(v);
85     }
86
87     void dfs_all() {
88         init();
89         for (int v = 0; v < n; v++) {
90             if (depth[v] == -1) {
91                 do_dfs_from(v);
92             }
93         }
94         assert((int) order.size() == n);
95     }
96 };

```

3.8 dfs-undigraph.cpp

```

1  template <typename T>
2  class dfs_undigraph : public undigraph<T> {
3  public:
4      using undigraph<T>::edges;
5      using undigraph<T>::g;
6      using undigraph<T>::n;
7
8      vector<int> pv;
9      vector<int> pe;

```

```

10     vector<int> order;
11     vector<int> pos;
12     vector<int> end;
13     vector<int> sz;
14     vector<int> root;
15     vector<int> depth;
16     vector<int> min_depth;
17     vector<T> dist;
18     vector<int> was;
19     int attempt;
20
21     dfs_undigraph(int _n) : undigraph<T>(_n) {
22     }
23
24     void init() {
25         pv = vector<int>(n, -1);
26         pe = vector<int>(n, -1);
27         order.clear();
28         pos = vector<int>(n, -1);
29         end = vector<int>(n, -1);
30         sz = vector<int>(n, 0);
31         root = vector<int>(n, -1);
32         depth = vector<int>(n, -1);
33         min_depth = vector<int>(n, -1);
34         dist = vector<T>(n);
35         was = vector<int>(n, -1);
36         attempt = 0;
37     }
38
39     void clear() {
40         pv.clear();
41         pe.clear();
42         order.clear();
43         pos.clear();
44         end.clear();
45         sz.clear();
46         root.clear();
47         depth.clear();
48         min_depth.clear();
49         dist.clear();
50         was.clear();
51     }

```

```

52
53 private:
54     void do_dfs(int v) {
55         was[v] = attempt;
56         pos[v] = (int) order.size();
57         order.push_back(v);
58         sz[v] = 1;
59         min_depth[v] = depth[v];
60         for (int id : g[v]) {
61             if (id == pe[v]) {
62                 continue;
63             }
64             auto &e = edges[id];
65             int to = e.from ^ e.to ^ v;
66             if (was[to] == attempt) {
67                 min_depth[v] = min(min_depth[v], depth[to]);
68                 continue;
69             }
70             depth[to] = depth[v] + 1;
71             dist[to] = dist[v] + e.cost;
72             pv[to] = v;
73             pe[to] = id;
74             root[to] = (root[v] != -1 ? root[v] : to);
75             do_dfs(to);
76             sz[v] += sz[to];
77             min_depth[v] = min(min_depth[v], min_depth[to]);
78         }
79         end[v] = (int) order.size() - 1;
80     }
81
82     void do_dfs_from(int v) {
83         ++attempt;
84         depth[v] = 0;
85         dist[v] = T{};
86         root[v] = v;
87         pv[v] = pe[v] = -1;
88         do_dfs(v);
89     }
90
91 public:
92     void dfs(int v, bool clear_order = true) {
93         if (pv.empty()) {

```

```

94             init();
95         } else {
96             if (clear_order) {
97                 order.clear();
98             }
99         }
100         do_dfs_from(v);
101     }
102
103     void dfs_all() {
104         init();
105         for (int v = 0; v < n; v++) {
106             if (depth[v] == -1) {
107                 do_dfs_from(v);
108             }
109         }
110         assert((int) order.size() == n);
111     }
112 };

```

3.9 digraph.cpp

```

1 template <typename T>
2 class digraph : public graph<T> {
3 public:
4     using graph<T>::edges;
5     using graph<T>::g;
6     using graph<T>::n;
7
8     digraph(int _n) : graph<T>(_n) {
9     }
10
11     int add(int from, int to, T cost = 1) {
12         assert(0 <= from && from < n && 0 <= to && to < n);
13         int id = (int) edges.size();
14         g[from].push_back(id);
15         edges.push_back({from, to, cost});
16         return id;
17     }
18
19     digraph<T> reverse() const {

```



```

20     digraph<T> rev(n);
21     for (auto &e : edges) {
22         rev.add(e.to, e.from, e.cost);
23     }
24     return rev;
25 }
26 };

```

3.10 dijkstra-set.cpp

```

1  template <typename T>
2  vector<T> dijkstra(const graph<T> &g, int start) {
3      assert(0 <= start && start < g.n);
4      vector<T> dist(g.n, numeric_limits<T>::max());
5      dist[start] = 0;
6      set<pair<T, int>> s;
7      s.emplace(dist[start], start);
8      while (!s.empty()) {
9          int i = s.begin()->second;
10         s.erase(s.begin());
11         for (int id : g.g[i]) {
12             auto &e = g.edges[id];
13             int to = e.from ^ e.to ^ i;
14             if (dist[i] + e.cost < dist[to]) {
15                 s.erase({dist[to], to});
16                 dist[to] = dist[i] + e.cost;
17                 s.emplace(dist[to], to);
18             }
19         }
20     }
21     return dist;
22     // returns numeric_limits<T>::max() if there's no path
23 }

```

3.11 dijkstra.cpp

```

1  template <typename T>
2  vector<T> dijkstra(const graph<T> &g, int start) {
3      assert(0 <= start && start < g.n);
4      vector<T> dist(g.n, numeric_limits<T>::max());
5      priority_queue<pair<T, int>, vector<pair<T, int>>, greater<pair<T, int

```

```

>>> s;
6  dist[start] = 0;
7  s.emplace(dist[start], start);
8  while (!s.empty()) {
9      T expected = s.top().first;
10     int i = s.top().second;
11     s.pop();
12     if (dist[i] != expected) {
13         continue;
14     }
15     for (int id : g.g[i]) {
16         auto &e = g.edges[id];
17         int to = e.from ^ e.to ^ i;
18         if (dist[i] + e.cost < dist[to]) {
19             dist[to] = dist[i] + e.cost;
20             s.emplace(dist[to], to);
21         }
22     }
23 }
24 return dist;
25 // returns numeric_limits<T>::max() if there's no path
26 }

```

3.12 dominators.cpp

```

1  template <typename T>
2  vector<int> find_dominators(const digraph<T> &g, int root) {
3      int n = g.n;
4      vector<int> pos(n, -1);
5      vector<int> order;
6      vector<int> parent(n, -1);
7      function<void(int)> dfs = [&g, &pos, &order, &parent, &dfs](int v) {
8          pos[v] = (int) order.size();
9          order.push_back(v);
10         for (int id : g.g[v]) {
11             auto &e = g.edges[id];
12             int u = e.to;
13             if (pos[u] == -1) {
14                 parent[u] = v;
15                 dfs(u);
16             }

```

```

17     }
18 };
19 dfs(root);
20 vector<int> p(n), best(n);
21 iota(p.begin(), p.end(), 0);
22 iota(best.begin(), best.end(), 0);
23 vector<int> sdom = pos;
24 function<int(int)> find_best = [&p, &best, &sdom, &find_best](int x) {
25     if (p[x] != x) {
26         int u = find_best(p[x]);
27         if (sdom[u] < sdom[best[x]]) {
28             best[x] = u;
29         }
30         p[x] = p[p[x]];
31     }
32     if (sdom[best[p[x]]] < sdom[best[x]]) {
33         best[x] = best[p[x]];
34     }
35     return best[x];
36 };
37 digraph<int> g_rev = g.reverse();
38 vector<int> idom(n, -1);
39 vector<int> link(n, 0);
40 vector<vector<int>> bucket(n);
41 for (int it = (int) order.size() - 1; it >= 0; it--) {
42     int w = order[it];
43     for (int id : g_rev.g[w]) {
44         auto &e = g_rev.edges[id];
45         int u = e.to;
46         if (pos[u] != -1) {
47             sdom[w] = min(sdom[w], sdom[find_best(u)]);
48         }
49     }
50     idom[w] = order[sdom[w]];
51     for (int u : bucket[w]) {
52         link[u] = find_best(u);
53     }
54     for (int id : g.g[w]) {
55         auto &e = g.edges[id];
56         int u = e.to;
57         if (parent[u] == w) {
58             p[u] = w;

```

```

59     }
60 }
61 bucket[order[sdom[w]]].push_back(w);
62 }
63 for (int it = 1; it < (int) order.size(); it++) {
64     int w = order[it];
65     idom[w] = idom[link[w]];
66 }
67 return idom;
68 // idom[i] -- immediate dominator for vertex i
69 }

```

3.13 eulerian.cpp

```

1 template <typename T>
2 vector<int> find_eulerian_path(const graph<T> &g, int &root) {
3     // in_deg and out_deg are fake for undigraph!
4     vector<int> in_deg(g.n, 0);
5     vector<int> out_deg(g.n, 0);
6     int cnt_edges = 0;
7     for (int id = 0; id < (int) g.edges.size(); id++) {
8         cnt_edges++;
9         auto &e = g.edges[id];
10        out_deg[e.from]++;
11        in_deg[e.to]++;
12    }
13    root = -1;
14    int odd = 0;
15    for (int i = 0; i < g.n; i++) {
16        if ((in_deg[i] + out_deg[i]) % 2 == 1) {
17            odd++;
18            if (root == -1 || out_deg[i] - in_deg[i] > out_deg[root] - in_deg[
19                root]) {
20                root = i;
21            }
22        }
23    }
24    if (odd > 2) {
25        return vector<int>();
26    }

```

```

27     if (root == -1) {
28         root = 0;
29         while (root < g.n && in_deg[root] + out_deg[root] == 0) {
30             root++;
31         }
32         if (root == g.n) {
33             // an empty path
34             root = 0;
35             return vector<int>();
36         }
37     }
38     vector<bool> used(g.edges.size(), false);
39     vector<int> ptr(g.n, 0);
40     vector<int> balance(g.n, 0);
41     vector<int> res(cnt_edges);
42     int stack_ptr = 0;
43     int write_ptr = cnt_edges;
44     int v = root;
45     while (true) {
46         bool found = false;
47         while (ptr[v] < (int) g.g[v].size()) {
48             int id = g.g[v][ptr[v]++];
49             if (used[id]) {
50                 continue;
51             }
52             used[id] = true;
53             res[stack_ptr++] = id;
54             auto &e = g.edges[id];
55             balance[v]++;
56             v ^= e.from ^ e.to;
57             balance[v]--;
58             found = true;
59             break;
60         }
61         if (!found) {
62             if (stack_ptr == 0) {
63                 break;
64             }
65             int id = res[--stack_ptr];
66             res[--write_ptr] = id;
67             auto &e = g.edges[id];
68             v ^= e.from ^ e.to;

```

```

69     }
70 }
71 int disbalance = 0;
72 for (int i = 0; i < g.n; i++) {
73     disbalance += abs(balance[i]);
74 }
75 if (write_ptr != 0 || disbalance > 2) {
76     root = -1;
77     return vector<int>();
78 }
79 return res;
80 // returns edge ids in the path (or the cycle if it exists)
81 // root == -1 if there is no path
82 // (or res.empty(), but this is also true when there are no edges)
83 }

```

3.14 forest.cpp

```

1  template <typename T>
2  class forest : public graph<T> {
3  public:
4      using graph<T>::edges;
5      using graph<T>::g;
6      using graph<T>::n;
7
8      forest(int _n) : graph<T>(_n) {
9      }
10
11     int add(int from, int to, T cost = 1) {
12         assert(0 <= from && from < n && 0 <= to && to < n);
13         int id = (int) edges.size();
14         assert(id < n - 1);
15         g[from].push_back(id);
16         g[to].push_back(id);
17         edges.push_back({from, to, cost});
18         return id;
19     }
20 };

```

3.15 graph.cpp

```

1  template <typename T>
2  class graph {
3  public:
4      struct edge {
5          int from;
6          int to;
7          T cost;
8      };
9
10     vector<edge> edges;
11     vector<vector<int>>> g;
12     int n;
13
14     graph(int _n) : n(_n) {
15         g.resize(n);
16     }
17
18     virtual int add(int from, int to, T cost) = 0;
19 };

```

3.16 hld-forest.cpp

```

1  template <typename T>
2  class hld_forest : public lca_forest<T> {
3  public:
4      using lca_forest<T>::edges;
5      using lca_forest<T>::g;
6      using lca_forest<T>::n;
7      using lca_forest<T>::pv;
8      using lca_forest<T>::sz;
9      using lca_forest<T>::pos;
10     using lca_forest<T>::order;
11     using lca_forest<T>::depth;
12     using lca_forest<T>::dfs;
13     using lca_forest<T>::dfs_all;
14     using lca_forest<T>::lca;
15     using lca_forest<T>::build_lca;
16
17     vector<int> head;
18     vector<int> visited;
19

```

```

20     hld_forest(int _n) : lca_forest<T>(_n) {
21         visited.resize(n);
22     }
23
24     void build_hld(const vector<int> &vs) {
25         for (int tries = 0; tries < 2; tries++) {
26             if (vs.empty()) {
27                 dfs_all();
28             } else {
29                 order.clear();
30                 for (int v : vs) {
31                     dfs(v, false);
32                 }
33                 assert((int) order.size() == n);
34             }
35             if (tries == 1) {
36                 break;
37             }
38             for (int i = 0; i < n; i++) {
39                 if (g[i].empty()) {
40                     continue;
41                 }
42                 int best = -1, bid = 0;
43                 for (int j = 0; j < (int) g[i].size(); j++) {
44                     int id = g[i][j];
45                     int v = edges[id].from ^ edges[id].to ^ i;
46                     if (pv[v] != i) {
47                         continue;
48                     }
49                     if (sz[v] > best) {
50                         best = sz[v];
51                         bid = j;
52                     }
53                 }
54                 swap(g[i][0], g[i][bid]);
55             }
56         }
57         build_lca();
58         head.resize(n);
59         for (int i = 0; i < n; i++) {
60             head[i] = i;
61         }

```

```

62     for (int i = 0; i < n - 1; i++) {
63         int x = order[i];
64         int y = order[i + 1];
65         if (pv[y] == x) {
66             head[y] = head[x];
67         }
68     }
69 }
70
71 void build_hld(int v) {
72     build_hld(vector<int>(1, v));
73 }
74
75 void build_hld_all() {
76     build_hld(vector<int>());
77 }
78
79 bool apply_on_path(int x, int y, bool with_lca, function<void(int,int,
80     bool)> f) {
81     // f(x, y, up): up -- whether this part of the path goes up
82     assert(!head.empty());
83     int z = lca(x, y);
84     if (z == -1) {
85         return false;
86     }
87     int v = x;
88     while (v != z) {
89         if (depth[head[v]] <= depth[z]) {
90             f(pos[z] + 1, pos[v], true);
91             break;
92         }
93         f(pos[head[v]], pos[v], true);
94         v = pv[head[v]];
95     }
96 }
97 if (with_lca) {
98     f(pos[z], pos[z], false);
99 }
100 {
101     int v = y;
102     int cnt_visited = 0;

```

```

103     while (v != z) {
104         if (depth[head[v]] <= depth[z]) {
105             f(pos[z] + 1, pos[v], false);
106             break;
107         }
108         visited[cnt_visited++] = v;
109         v = pv[head[v]];
110     }
111     for (int at = cnt_visited - 1; at >= 0; at--) {
112         v = visited[at];
113         f(pos[head[v]], pos[v], false);
114     }
115 }
116 return true;
117 }
118 };

```

3.17 lca-forest.cpp

```

1 template <typename T>
2 class lca_forest : public dfs_forest<T> {
3 public:
4     using dfs_forest<T>::edges;
5     using dfs_forest<T>::g;
6     using dfs_forest<T>::n;
7     using dfs_forest<T>::pv;
8     using dfs_forest<T>::pos;
9     using dfs_forest<T>::end;
10    using dfs_forest<T>::depth;
11
12    int h;
13    vector<vector<int>> pr;
14
15    lca_forest(int _n) : dfs_forest<T>(_n) {
16    }
17
18    inline void build_lca() {
19        assert(!pv.empty());
20        int max_depth = 0;
21        for (int i = 0; i < n; i++) {
22            max_depth = max(max_depth, depth[i]);

```

```

23     }
24     h = 1;
25     while ((1 << h) <= max_depth) {
26         h++;
27     }
28     pr.resize(n);
29     for (int i = 0; i < n; i++) {
30         pr[i].resize(h);
31         pr[i][0] = pv[i];
32     }
33     for (int j = 1; j < h; j++) {
34         for (int i = 0; i < n; i++) {
35             pr[i][j] = (pr[i][j - 1] == -1 ? -1 : pr[pr[i][j - 1]][j - 1]);
36         }
37     }
38 }
39
40 inline bool anc(int x, int y) {
41     return (pos[x] <= pos[y] && end[y] <= end[x]);
42 }
43
44 inline int go_up(int x, int up) {
45     assert(!pr.empty());
46     up = min(up, (1 << h) - 1);
47     for (int j = h - 1; j >= 0; j--) {
48         if (up & (1 << j)) {
49             x = pr[x][j];
50             if (x == -1) {
51                 break;
52             }
53         }
54     }
55     return x;
56 }
57
58 inline int lca(int x, int y) {
59     assert(!pr.empty());
60     if (anc(x, y)) {
61         return x;
62     }
63     if (anc(y, x)) {
64         return y;

```

```

65     }
66     for (int j = h - 1; j >= 0; j--) {
67         if (pr[x][j] != -1 && !anc(pr[x][j], y)) {
68             x = pr[x][j];
69         }
70     }
71     return pr[x][0];
72 }
73 };

```

3.18 mst.cpp

```

1  template <typename T>
2  vector<int> find_mst(const undigraph<T> &g, T &ans) {
3      vector<int> order(g.edges.size());
4      iota(order.begin(), order.end(), 0);
5      sort(order.begin(), order.end(), [&g](int a, int b) {
6          return g.edges[a].cost < g.edges[b].cost;
7      });
8      dsu d(g.n);
9      vector<int> ans_list;
10     ans = 0;
11     for (int id : order) {
12         auto &e = g.edges[id];
13         if (d.get(e.from) != d.get(e.to)) {
14             d.unite(e.from, e.to);
15             ans_list.push_back(id);
16             ans += e.cost;
17         }
18     }
19     return ans_list;
20     // returns edge ids of minimum "spanning" forest
21 }

```

3.19 scc.cpp

```

1  template <typename T>
2  vector<int> find_scc(const digraph<T> &g, int &cnt) {
3      digraph<T> g_rev = g.reverse();
4      vector<int> order;
5      vector<bool> was(g.n, false);

```

```

6  function<void(int)> dfs1 = [&](int v) {
7      was[v] = true;
8      for (int id : g.g[v]) {
9          auto &e = g.edges[id];
10         int to = e.to;
11         if (!was[to]) {
12             dfs1(to);
13         }
14     }
15     order.push_back(v);
16 };
17 for (int i = 0; i < g.n; i++) {
18     if (!was[i]) {
19         dfs1(i);
20     }
21 }
22 vector<int> c(g.n, -1);
23 function<void(int)> dfs2 = [&](int v) {
24     for (int id : g_rev.g[v]) {
25         auto &e = g_rev.edges[id];
26         int to = e.to;
27         if (c[to] == -1) {
28             c[to] = c[v];
29             dfs2(to);
30         }
31     }
32 };
33 cnt = 0;
34 for (int id = g.n - 1; id >= 0; id--) {
35     int i = order[id];
36     if (c[i] != -1) {
37         continue;
38     }
39     c[i] = cnt++;
40     dfs2(i);
41 }
42 return c;
43 // c[i] <= c[j] for every edge i -> j
44 }

```

3.20 topsort.cpp

```

1  template <typename T>
2  vector<int> find_topsort(const digraph<T> &g) {
3      vector<int> deg(g.n, 0);
4      for (int id = 0; id < (int) g.edges.size(); id++) {
5          deg[g.edges[id].to]++;
6      }
7      vector<int> x;
8      for (int i = 0; i < g.n; i++) {
9          if (deg[i] == 0) {
10             x.push_back(i);
11         }
12     }
13     for (int ptr = 0; ptr < (int) x.size(); ptr++) {
14         int i = x[ptr];
15         for (int id : g.g[i]) {
16             auto &e = g.edges[id];
17             int to = e.to;
18             if (--deg[to] == 0) {
19                 x.push_back(to);
20             }
21         }
22     }
23     if ((int) x.size() != g.n) {
24         return vector<int>();
25     }
26     return x;
27 }

```

3.21 twosat.cpp

```

1  class twosat {
2  public:
3      digraph<int> g;
4      int n;
5
6      twosat(int _n) : g(digraph<int>(2 * _n)), n(_n) {
7      }
8
9      // (v[x] == value_x)
10     inline void add(int x, int value_x) {

```

```

11     assert(0 <= x && x < n);
12     assert(0 <= value_x && value_x <= 1);
13     g.add(2 * x + (value_x ^ 1), 2 * x + value_x);
14 }
15
16 // (v[x] == value_x || v[y] == value_y)
17 inline void add(int x, int value_x, int y, int value_y) {
18     assert(0 <= x && x < n && 0 <= y && y < n);
19     assert(0 <= value_x && value_x <= 1 && 0 <= value_y && value_y <= 1);
20     g.add(2 * x + (value_x ^ 1), 2 * y + value_y);
21     g.add(2 * y + (value_y ^ 1), 2 * x + value_x);
22 }
23
24 inline vector<int> solve() {
25     int cnt;
26     vector<int> c = find_scc(g, cnt);
27     vector<int> res(n);
28     for (int i = 0; i < n; i++) {
29         if (c[2 * i] == c[2 * i + 1]) {
30             return vector<int>();
31         }
32         res[i] = (c[2 * i] < c[2 * i + 1]);
33     }
34     return res;
35 }
36 };

```

3.22 undigraph.cpp

```

1 template <typename T>
2 class undigraph : public graph<T> {
3 public:
4     using graph<T>::edges;
5     using graph<T>::g;
6     using graph<T>::n;
7
8     undigraph(int _n) : graph<T>(_n) {
9     }
10
11     int add(int from, int to, T cost = 1) {
12         assert(0 <= from && from < n && 0 <= to && to < n);

```

```

13         int id = (int) edges.size();
14         g[from].push_back(id);
15         g[to].push_back(id);
16         edges.push_back({from, to, cost});
17         return id;
18     }
19 };

```

4 misc

4.1 debug.cpp

```

1 template <typename A, typename B>
2 string to_string(pair<A, B> p);
3
4 template <typename A, typename B, typename C>
5 string to_string(tuple<A, B, C> p);
6
7 template <typename A, typename B, typename C, typename D>
8 string to_string(tuple<A, B, C, D> p);
9
10 string to_string(const string& s) {
11     return '"' + s + '"';
12 }
13
14 string to_string(const char* s) {
15     return to_string((string) s);
16 }
17
18 string to_string(bool b) {
19     return (b ? "true" : "false");
20 }
21
22 string to_string(vector<bool> v) {
23     bool first = true;
24     string res = "{";
25     for (int i = 0; i < static_cast<int>(v.size()); i++) {
26         if (!first) {
27             res += ", ";
28         }
29         first = false;

```



```

30     res += to_string(v[i]);
31 }
32 res += "}";
33 return res;
34 }
35
36 template <size_t N>
37 string to_string(bitset<N> v) {
38     string res = "";
39     for (size_t i = 0; i < N; i++) {
40         res += static_cast<char>('0' + v[i]);
41     }
42     return res;
43 }
44
45 template <typename A>
46 string to_string(A v) {
47     bool first = true;
48     string res = "{";
49     for (const auto &x : v) {
50         if (!first) {
51             res += ", ";
52         }
53         first = false;
54         res += to_string(x);
55     }
56     res += "}";
57     return res;
58 }
59
60 template <typename A, typename B>
61 string to_string(pair<A, B> p) {
62     return "(" + to_string(p.first) + ", " + to_string(p.second) + ")";
63 }
64
65 template <typename A, typename B, typename C>
66 string to_string(tuple<A, B, C> p) {
67     return "(" + to_string(get<0>(p)) + ", " + to_string(get<1>(p)) + ", " +
68         to_string(get<2>(p)) + ")";
69 }
70
71 template <typename A, typename B, typename C, typename D>

```

```

71 string to_string(tuple<A, B, C, D> p) {
72     return "(" + to_string(get<0>(p)) + ", " + to_string(get<1>(p)) + ", " +
73         to_string(get<2>(p)) + ", " + to_string(get<3>(p)) + ")";
74 }
75
76 void debug_out() { cerr << endl; }
77
78 template <typename Head, typename... Tail>
79 void debug_out(Head H, Tail... T) {
80     cerr << " " << to_string(H);
81     debug_out(T...);
82 }
83
84 #ifdef LOCAL
85 #define debug(...) cerr << "[" << #__VA_ARGS__ << "]:", debug_out(
86     __VA_ARGS__)
87 #else
88 #define debug(...) 42
89 #endif

```

4.2 fastinput.cpp

```

1 static struct FastInput {
2     static constexpr int BUF_SIZE = 1 << 20;
3     char buf[BUF_SIZE];
4     size_t chars_read = 0;
5     size_t buf_pos = 0;
6     FILE *in = stdin;
7     char cur = 0;
8
9     inline char get_char() {
10         if (buf_pos >= chars_read) {
11             chars_read = fread(buf, 1, BUF_SIZE, in);
12             buf_pos = 0;
13             buf[0] = (chars_read == 0 ? -1 : buf[0]);
14         }
15         return cur = buf[buf_pos++];
16     }
17
18     inline void tie(int) {}
19 }

```

```

20 inline explicit operator bool() {
21     return cur != -1;
22 }
23
24 inline static bool is_blank(char c) {
25     return c <= ' ';
26 }
27
28 inline bool skip_blanks() {
29     while (is_blank(cur) && cur != -1) {
30         get_char();
31     }
32     return cur != -1;
33 }
34
35 inline FastInput& operator>>(char& c) {
36     skip_blanks();
37     c = cur;
38     return *this;
39 }
40
41 inline FastInput& operator>>(string& s) {
42     if (skip_blanks()) {
43         s.clear();
44         do {
45             s += cur;
46         } while (!is_blank(get_char()));
47     }
48     return *this;
49 }
50
51 template <typename T>
52 inline FastInput& read_integer(T& n) {
53     // unsafe, doesn't check that characters are actually digits
54     n = 0;
55     if (skip_blanks()) {
56         int sign = +1;
57         if (cur == '-') {
58             sign = -1;
59             get_char();
60         }
61         do {

```

```

62             n += n * (n << 3) + cur - '0';
63         } while (!is_blank(get_char()));
64         n *= sign;
65     }
66     return *this;
67 }
68
69 template <typename T>
70 inline typename enable_if<is_integral<T>::value, FastInput&>::type
71     operator>>(T& n) {
72     return read_integer(n);
73 }
74
75 #if !defined(_WIN32) || defined(_WIN64)
76 inline FastInput& operator>>(__int128& n) {
77     return read_integer(n);
78 }
79 #endif
80
81 template <typename T>
82 inline typename enable_if<is_floating_point<T>::value, FastInput&>::type
83     operator>>(T& n) {
84     // not sure if really fast, for compatibility only
85     n = 0;
86     if (skip_blanks()) {
87         string s;
88         (*this) >> s;
89         sscanf(s.c_str(), "%lf", &n);
90     }
91     return *this;
92 } fast_input;
93 #define cin fast_input

```

4.3 fastoutput.cpp

```

1 static struct FastOutput {
2     static constexpr int BUF_SIZE = 1 << 20;
3     char buf[BUF_SIZE];
4     size_t buf_pos = 0;

```

```

5  static constexpr int TMP_SIZE = 1 << 20;
6  char tmp[TMP_SIZE];
7  FILE *out = stdout;
8
9  inline void put_char(char c) {
10     buf[buf_pos++] = c;
11     if (buf_pos == BUF_SIZE) {
12         fwrite(buf, 1, buf_pos, out);
13         buf_pos = 0;
14     }
15 }
16
17 ~FastOutput() {
18     fwrite(buf, 1, buf_pos, out);
19 }
20
21 inline FastOutput& operator<<(char c) {
22     put_char(c);
23     return *this;
24 }
25
26 inline FastOutput& operator<<(const char* s) {
27     while (*s) {
28         put_char(*s++);
29     }
30     return *this;
31 }
32
33 inline FastOutput& operator<<(const string& s) {
34     for (int i = 0; i < (int) s.size(); i++) {
35         put_char(s[i]);
36     }
37     return *this;
38 }
39
40 template <typename T>
41 inline char* integer_to_string(T n) {
42     // beware of TMP_SIZE
43     char* p = tmp + TMP_SIZE - 1;
44     if (n == 0) {
45         *--p = '0';
46     } else {

```

```

47         bool is_negative = false;
48         if (n < 0) {
49             is_negative = true;
50             n = -n;
51         }
52         while (n > 0) {
53             *--p = (char) ('0' + n % 10);
54             n /= 10;
55         }
56         if (is_negative) {
57             *--p = '-';
58         }
59     }
60     return p;
61 }
62
63 template <typename T>
64 inline typename enable_if<is_integral<T>::value, char*>::type stringify(
65     T n) {
66     return integer_to_string(n);
67 }
68
69 #if !defined(_WIN32) || defined(_WIN64)
70 inline char* stringify(__int128 n) {
71     return integer_to_string(n);
72 }
73 #endif
74
75 template <typename T>
76 inline typename enable_if<is_floating_point<T>::value, char*>::type
77     stringify(T n) {
78     sprintf(tmp, "%.17f", n);
79     return tmp;
80 }
81
82 template <typename T>
83 inline FastOutput& operator<<(const T& n) {
84     auto p = stringify(n);
85     for (; *p != 0; p++) {
86         put_char(*p);
87     }
88     return *this;

```

```

87     }
88 } fast_output;
89
90 #define cout fast_output

```

4.4 lis.cpp

```

1  template<typename T>
2  int lis(const vector<T>& a) {
3      vector<T> u;
4      for (const T& x : a) {
5          auto it = lower_bound(u.begin(), u.end(), x);
6          if (it == u.end()) {
7              u.push_back(x);
8          } else {
9              *it = x;
10         }
11     }
12     return (int) u.size();
13 }

```

4.5 radix.cpp

```

1  namespace radix {
2
3  vector<int> p(65537);
4
5  template<typename T>
6  void SortShift(vector<T>& a, vector<T>& new_a, int shift) {
7      assert(a.size() == new_a.size());
8      int n = static_cast<int>(a.size());
9      fill(p.begin(), p.end(), 0);
10     for (int i = 0; i < n; i++) p[1 + ((a[i] >> shift) & 0xffff)]++;
11     for (int i = 1; i <= 65536; i++) p[i] += p[i - 1];
12     for (int i = 0; i < n; i++) new_a[p[(a[i] >> shift) & 0xffff]++] = a[i];
13 }
14
15 void Sort(vector<int32_t>& a) {
16     constexpr int32_t flip = static_cast<int32_t>(1) << 31;
17     for (auto& aa : a) aa ^= flip;
18     vector<int32_t> b(a.size());

```

```

19     SortShift(a, b, 0);
20     SortShift(b, a, 16);
21     for (auto& aa : a) aa ^= flip;
22 }
23
24 void Sort(vector<uint32_t>& a) {
25     vector<uint32_t> b(a.size());
26     SortShift(a, b, 0);
27     SortShift(b, a, 16);
28 }
29
30 void Sort(vector<int64_t>& a) {
31     constexpr int64_t flip = static_cast<int64_t>(1) << 63;
32     for (auto& aa : a) aa ^= flip;
33     vector<int64_t> b(a.size());
34     SortShift(a, b, 0);
35     SortShift(b, a, 16);
36     SortShift(a, b, 32);
37     SortShift(b, a, 48);
38     for (auto& aa : a) aa ^= flip;
39 }
40
41 void Sort(vector<uint64_t>& a) {
42     vector<uint64_t> b(a.size());
43     SortShift(a, b, 0);
44     SortShift(b, a, 16);
45     SortShift(a, b, 32);
46     SortShift(b, a, 48);
47 }
48
49 } // namespace radix

```

4.6 rng.cpp

```

1  mt19937_64 rng((unsigned int) chrono::steady_clock::now().time_since_epoch
    ().count());

```

5 numeric

5.1 bm.cpp

```
1 template <typename T>
2 vector<T> BM(vector<T> a) {
3     vector<T> p = {1};
4     vector<T> q = {1};
5     int l = 0;
6     for (int r = 1; r <= (int) a.size(); r++) {
7         T delta = 0;
8         for (int j = 0; j <= l; j++) {
9             delta += a[r - 1 - j] * p[j];
10        }
11        q.insert(q.begin(), 0);
12        if (delta != 0) {
13            vector<T> t = p;
14            if (q.size() > t.size()) {
15                t.resize(q.size());
16            }
17            for (int i = 0; i < (int) q.size(); i++) {
18                t[i] -= delta * q[i];
19            }
20            if (2 * l <= r - 1) {
21                q = p;
22                T od = 1 / delta;
23                for (T& x : q) {
24                    x *= od;
25                }
26                l = r - 1;
27            }
28            swap(p, t);
29        }
30    }
31    assert((int) p.size() == l + 1);
32    // assert(l * 2 + 30 < (int) a.size());
33    reverse(p.begin(), p.end());
34    return p;
35 }
```

5.2 extgcd.cpp

```
1 template<typename T>
2 T extgcd(T a, T b, T &x, T &y) {
3     if (a == 0) {
4         x = 0;
5         y = 1;
6         return b;
7     }
8     T p = b / a;
9     T g = extgcd(b - p * a, a, y, x);
10    x -= p * y;
11    return g;
12 }
13
14 template<typename T>
15 bool diophantine(T a, T b, T c, T &x, T &y, T &g) {
16     if (a == 0 && b == 0) {
17         if (c == 0) {
18             x = y = g = 0;
19             return true;
20         }
21         return false;
22     }
23     if (a == 0) {
24         if (c % b == 0) {
25             x = 0;
26             y = c / b;
27             g = abs(b);
28             return true;
29         }
30         return false;
31     }
32     if (b == 0) {
33         if (c % a == 0) {
34             x = c / a;
35             y = 0;
36             g = abs(a);
37             return true;
38         }
39         return false;
40     }
```

```

41 g = extgcd(a, b, x, y);
42 if (c % g != 0) {
43     return false;
44 }
45 T dx = c / a;
46 c -= dx * a;
47 T dy = c / b;
48 c -= dy * b;
49 x = dx + (T) ((__int128) x * (c / g) % b);
50 y = dy + (T) ((__int128) y * (c / g) % a);
51 g = abs(g);
52 return true;
53 // |x|, |y| <= max(|a|, |b|, |c|) [tested]
54 }
55
56 bool crt(long long k1, long long m1, long long k2, long long m2, long long
    &k, long long &m) {
57     k1 %= m1;
58     if (k1 < 0) k1 += m1;
59     k2 %= m2;
60     if (k2 < 0) k2 += m2;
61     long long x, y, g;
62     if (!diophantine(m1, -m2, k2 - k1, x, y, g)) {
63         return false;
64     }
65     long long dx = m2 / g;
66     long long delta = x / dx - (x % dx < 0);
67     k = m1 * (x - dx * delta) + k1;
68     m = m1 / g * m2;
69     assert(0 <= k && k < m);
70     return true;
71 }
72
73 // for distinct prime modulus
74 template <typename T>
75 void crt_garner(const vector<int>& p, const vector<int>& a, T& res) {
76     assert(p.size() == a.size());
77     auto inverse = [&](int q, int m) {
78         q %= m;
79         if (q < 0) q += m;
80         int b = m, u = 0, v = 1;
81         while (q) {

```

```

82             int t = b / q;
83             b -= t * q; swap(q, b);
84             u -= t * v; swap(u, v);
85         }
86         assert(b == 1);
87         if (u < 0) u += m;
88         return u;
89     };
90     vector<int> x(p.size());
91     for (int i = 0; i < (int) p.size(); i++) {
92         assert(0 <= a[i] && a[i] < p[i]);
93         x[i] = a[i];
94         for (int j = 0; j < i; j++) {
95             x[i] = (int) ((long long) (x[i] - x[j]) * inverse(p[j], p[i]) % p[i]
                );
96             if (x[i] < 0) x[i] += p[i];
97         }
98     }
99     res = 0;
100     for (int i = (int) p.size() - 1; i >= 0; i--) {
101         res = res * p[i] + x[i];
102     }
103 }

```

5.3 factorizer.cpp

```

1 namespace factorizer {
2
3 template <typename T>
4 struct FactorizerVarMod { static T value; };
5 template <typename T>
6 T FactorizerVarMod<T>::value;
7
8 template <typename T>
9 bool IsPrime(T n, const vector<T>& bases) {
10     if (n < 2) {
11         return false;
12     }
13     vector<T> small_primes = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
14     for (const T& x : small_primes) {
15         if (n % x == 0) {

```

```

16     return n == x;
17 }
18 }
19 if (n < 31 * 31) {
20     return true;
21 }
22 int s = 0;
23 T d = n - 1;
24 while ((d & 1) == 0) {
25     d >>= 1;
26     s++;
27 }
28 FactorizerVarMod<T>::value = n;
29 for (const T& a : bases) {
30     if (a % n == 0) {
31         continue;
32     }
33     Modular<FactorizerVarMod<T>> cur = a;
34     cur = power(cur, d);
35     if (cur == 1) {
36         continue;
37     }
38     bool witness = true;
39     for (int r = 0; r < s; r++) {
40         if (cur == n - 1) {
41             witness = false;
42             break;
43         }
44         cur *= cur;
45     }
46     if (witness) {
47         return false;
48     }
49 }
50 return true;
51 }
52
53 bool IsPrime(int64_t n) {
54     return IsPrime(n, {2, 325, 9375, 28178, 450775, 9780504, 1795265022});
55 }
56
57 bool IsPrime(int32_t n) {

```

```

58     return IsPrime(n, {2, 7, 61});
59 }
60
61 // but if you really need uint64_t version...
62 /*
63 bool IsPrime(uint64_t n) {
64     if (n < 2) {
65         return false;
66     }
67     vector<uint32_t> small_primes = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
68     for (uint32_t x : small_primes) {
69         if (n == x) {
70             return true;
71         }
72         if (n % x == 0) {
73             return false;
74         }
75     }
76     if (n < 31 * 31) {
77         return true;
78     }
79     uint32_t s = __builtin_ctzll(n - 1);
80     uint64_t d = (n - 1) >> s;
81     function<bool(uint64_t)> witness = [&n, &s, &d](uint64_t a) {
82         uint64_t cur = 1, p = d;
83         while (p > 0) {
84             if (p & 1) {
85                 cur = (__uint128_t) cur * a % n;
86             }
87             a = (__uint128_t) a * a % n;
88             p >>= 1;
89         }
90         if (cur == 1) {
91             return false;
92         }
93         for (uint32_t r = 0; r < s; r++) {
94             if (cur == n - 1) {
95                 return false;
96             }
97             cur = (__uint128_t) cur * cur % n;
98         }
99         return true;

```

```

100     };
101     vector<uint64_t> bases_64bit = {2, 325, 9375, 28178, 450775, 9780504,
102         1795265022};
103     for (uint64_t a : bases_64bit) {
104         if (a % n == 0) {
105             return true;
106         }
107         if (witness(a)) {
108             return false;
109         }
110     }
111     return true;
112 }
113
114 vector<int> least = {0, 1};
115 vector<int> primes;
116 int precalculated = 1;
117
118 void RunLinearSieve(int n) {
119     n = max(n, 1);
120     least.assign(n + 1, 0);
121     primes.clear();
122     for (int i = 2; i <= n; i++) {
123         if (least[i] == 0) {
124             least[i] = i;
125             primes.push_back(i);
126         }
127         for (int x : primes) {
128             if (x > least[i] || i * x > n) {
129                 break;
130             }
131             least[i * x] = x;
132         }
133     }
134     precalculated = n;
135 }
136
137 void RunSlowSieve(int n) {
138     n = max(n, 1);
139     least.assign(n + 1, 0);
140     for (int i = 2; i * i <= n; i++) {

```

```

141         if (least[i] == 0) {
142             for (int j = i * i; j <= n; j += i) {
143                 if (least[j] == 0) {
144                     least[j] = i;
145                 }
146             }
147         }
148     }
149     primes.clear();
150     for (int i = 2; i <= n; i++) {
151         if (least[i] == 0) {
152             least[i] = i;
153             primes.push_back(i);
154         }
155     }
156     precalculated = n;
157 }
158
159 void RunSieve(int n) {
160     RunLinearSieve(n);
161 }
162
163 template <typename T>
164 vector<pair<T, int>> MergeFactors(const vector<pair<T, int>>& a, const
165     vector<pair<T, int>>& b) {
166     vector<pair<T, int>> c;
167     int i = 0;
168     int j = 0;
169     while (i < (int) a.size() || j < (int) b.size()) {
170         if (i < (int) a.size() && j < (int) b.size() && a[i].first == b[j].
171             first) {
172             c.emplace_back(a[i].first, a[i].second + b[j].second);
173             ++i;
174             ++j;
175             continue;
176         }
177         if (j == (int) b.size() || (i < (int) a.size() && a[i].first < b[j].
178             first)) {
179             c.push_back(a[i++]);
180         } else {
181             c.push_back(b[j++]);
182         }
183     }

```



```

180     }
181     return c;
182 }
183
184 template <typename T>
185 vector<pair<T, int>> RhoC(const T& n, const T& c) {
186     if (n <= 1) {
187         return {};
188     }
189     if ((n & 1) == 0) {
190         return MergeFactors({{2, 1}}, RhoC(n / 2, c));
191     }
192     if (IsPrime(n)) {
193         return {{n, 1}};
194     }
195     FactorizerVarMod<T>::value = n;
196     Modular<FactorizerVarMod<T>> x = 2;
197     Modular<FactorizerVarMod<T>> saved = 2;
198     T power = 1;
199     T lam = 1;
200     while (true) {
201         x = x * x + c;
202         T g = __gcd((x - saved)(), n);
203         if (g != 1) {
204             return MergeFactors(RhoC(g, c + 1), RhoC(n / g, c + 1));
205         }
206         if (power == lam) {
207             saved = x;
208             power <= 1;
209             lam = 0;
210         }
211         lam++;
212     }
213     return {};
214 }
215
216 template <typename T>
217 vector<pair<T, int>> Rho(const T& n) {
218     return RhoC(n, static_cast<T>(1));
219 }
220
221 template <typename T>

```

```

222 vector<pair<T, int>> Factorize(T x) {
223     if (x <= 1) {
224         return {};
225     }
226     if (x <= precalculated) {
227         vector<pair<T, int>> ret;
228         while (x > 1) {
229             if (!ret.empty() && ret.back().first == least[x]) {
230                 ret.back().second++;
231             } else {
232                 ret.emplace_back(least[x], 1);
233             }
234             x /= least[x];
235         }
236         return ret;
237     }
238     if (x <= static_cast<int64_t>(precalculated) * precalculated) {
239         vector<pair<T, int>> ret;
240         if (!IsPrime(x)) {
241             for (T i : primes) {
242                 T t = x / i;
243                 if (i > t) {
244                     break;
245                 }
246                 if (x == t * i) {
247                     int cnt = 0;
248                     while (x % i == 0) {
249                         x /= i;
250                         cnt++;
251                     }
252                     ret.emplace_back(i, cnt);
253                     if (IsPrime(x)) {
254                         break;
255                     }
256                 }
257             }
258         }
259         if (x > 1) {
260             ret.emplace_back(x, 1);
261         }
262         return ret;
263     }

```

```

264     return Rho(x);
265 }
266
267 template <typename T>
268 vector<T> BuildDivisorsFromFactors(const vector<pair<T, int>>& factors) {
269     vector<T> divisors = {1};
270     for (auto& p : factors) {
271         int sz = (int) divisors.size();
272         for (int i = 0; i < sz; i++) {
273             T cur = divisors[i];
274             for (int j = 0; j < p.second; j++) {
275                 cur *= p.first;
276                 divisors.push_back(cur);
277             }
278         }
279     }
280     sort(divisors.begin(), divisors.end());
281     return divisors;
282 }
283
284 } // namespace factorizer

```

5.4 fft.cpp

```

1 // make it understandable one day...
2 namespace fft {
3
4 typedef double dbl;
5
6 struct num {
7     dbl x, y;
8     num() { x = y = 0; }
9     num(dbl x_, dbl y_) : x(x_), y(y_) {}
10 };
11
12 inline num operator+(num a, num b) { return num(a.x + b.x, a.y + b.y); }
13 inline num operator-(num a, num b) { return num(a.x - b.x, a.y - b.y); }
14 inline num operator*(num a, num b) { return num(a.x * b.x - a.y * b.y, a.x
    * b.y + a.y * b.x); }
15 inline num conj(num a) { return num(a.x, -a.y); }
16

```

```

17 int base = 1;
18 vector<num> roots = {{0, 0}, {1, 0}};
19 vector<int> rev = {0, 1};
20
21 const dbl PI = static_cast<dbl>(acosl(-1.0));
22
23 void ensure_base(int nbase) {
24     if (nbase <= base) {
25         return;
26     }
27     rev.resize(1 << nbase);
28     for (int i = 0; i < (1 << nbase); i++) {
29         rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (nbase - 1));
30     }
31     roots.resize(1 << nbase);
32     while (base < nbase) {
33         dbl angle = 2 * PI / (1 << (base + 1));
34         // num z(cos(angle), sin(angle));
35         for (int i = 1 << (base - 1); i < (1 << base); i++) {
36             roots[i << 1] = roots[i];
37             // roots[(i << 1) + 1] = roots[i] * z;
38             dbl angle_i = angle * (2 * i + 1 - (1 << base));
39             roots[(i << 1) + 1] = num(cos(angle_i), sin(angle_i));
40         }
41         base++;
42     }
43 }
44
45 void fft(vector<num>& a, int n = -1) {
46     if (n == -1) {
47         n = (int) a.size();
48     }
49     assert((n & (n - 1)) == 0);
50     int zeros = __builtin_ctz(n);
51     ensure_base(zeros);
52     int shift = base - zeros;
53     for (int i = 0; i < n; i++) {
54         if (i < (rev[i] >> shift)) {
55             swap(a[i], a[rev[i] >> shift]);
56         }
57     }
58     for (int k = 1; k < n; k <= 1) {

```

```

59     for (int i = 0; i < n; i += 2 * k) {
60         for (int j = 0; j < k; j++) {
61             num z = a[i + j + k] * roots[j + k];
62             a[i + j + k] = a[i + j] - z;
63             a[i + j] = a[i + j] + z;
64         }
65     }
66 }
67 }
68
69 vector<num> fa, fb;
70
71 vector<int64_t> square(const vector<int>& a) {
72     if (a.empty()) {
73         return {};
74     }
75     int need = (int) a.size() + (int) a.size() - 1;
76     int nbase = 1;
77     while ((1 << nbase) < need) nbase++;
78     ensure_base(nbase);
79     int sz = 1 << nbase;
80     if ((sz >> 1) > (int) fa.size()) {
81         fa.resize(sz >> 1);
82     }
83     for (int i = 0; i < (sz >> 1); i++) {
84         int x = (2 * i < (int) a.size() ? a[2 * i] : 0);
85         int y = (2 * i + 1 < (int) a.size() ? a[2 * i + 1] : 0);
86         fa[i] = num(x, y);
87     }
88     fft(fa, sz >> 1);
89     num r(1.0 / (sz >> 1), 0.0);
90     for (int i = 0; i <= (sz >> 2); i++) {
91         int j = ((sz >> 1) - i) & ((sz >> 1) - 1);
92         num fe = (fa[i] + conj(fa[j])) * num(0.5, 0);
93         num fo = (fa[i] - conj(fa[j])) * num(0, -0.5);
94         num aux = fe * fe + fo * fo * roots[(sz >> 1) + i] * roots[(sz >> 1) +
95             i];
96         num tmp = fe * fo;
97         fa[i] = r * (conj(aux) + num(0, 2) * conj(tmp));
98         fa[j] = r * (aux + num(0, 2) * tmp);
99     }
100     fft(fa, sz >> 1);

```

```

100     vector<int64_t> res(need);
101     for (int i = 0; i < need; i++) {
102         res[i] = llround(i % 2 == 0 ? fa[i >> 1].x : fa[i >> 1].y);
103     }
104     return res;
105 }
106
107 vector<int64_t> multiply(const vector<int>& a, const vector<int>& b) {
108     if (a.empty() || b.empty()) {
109         return {};
110     }
111     if (a == b) {
112         return square(a);
113     }
114     int need = (int) a.size() + (int) b.size() - 1;
115     int nbase = 1;
116     while ((1 << nbase) < need) nbase++;
117     ensure_base(nbase);
118     int sz = 1 << nbase;
119     if (sz > (int) fa.size()) {
120         fa.resize(sz);
121     }
122     for (int i = 0; i < sz; i++) {
123         int x = (i < (int) a.size() ? a[i] : 0);
124         int y = (i < (int) b.size() ? b[i] : 0);
125         fa[i] = num(x, y);
126     }
127     fft(fa, sz);
128     num r(0, -0.25 / (sz >> 1));
129     for (int i = 0; i <= (sz >> 1); i++) {
130         int j = (sz - i) & (sz - 1);
131         num z = (fa[j] * fa[j] - conj(fa[i] * fa[i])) * r;
132         fa[j] = (fa[i] * fa[i] - conj(fa[j] * fa[j])) * r;
133         fa[i] = z;
134     }
135     for (int i = 0; i < (sz >> 1); i++) {
136         num A0 = (fa[i] + fa[i + (sz >> 1)]) * num(0.5, 0);
137         num A1 = (fa[i] - fa[i + (sz >> 1)]) * num(0.5, 0) * roots[(sz >> 1) +
138             i];
139         fa[i] = A0 + A1 * num(0, 1);
140     }
141     fft(fa, sz >> 1);

```

```

141     vector<int64_t> res(need);
142     for (int i = 0; i < need; i++) {
143         res[i] = llround(i % 2 == 0 ? fa[i >> 1].x : fa[i >> 1].y);
144     }
145     return res;
146 }
147
148 vector<int> multiply_mod(const vector<int>& a, const vector<int>& b, int m
    ) {
149     if (a.empty() || b.empty()) {
150         return {};
151     }
152     int eq = (a.size() == b.size() && a == b);
153     int need = (int) a.size() + (int) b.size() - 1;
154     int nbase = 0;
155     while ((1 << nbase) < need) nbase++;
156     ensure_base(nbase);
157     int sz = 1 << nbase;
158     if (sz > (int) fa.size()) {
159         fa.resize(sz);
160     }
161     for (int i = 0; i < (int) a.size(); i++) {
162         int x = (a[i] % m + m) % m;
163         fa[i] = num(x & ((1 << 15) - 1), x >> 15);
164     }
165     fill(fa.begin() + a.size(), fa.begin() + sz, num {0, 0});
166     fft(fa, sz);
167     if (sz > (int) fb.size()) {
168         fb.resize(sz);
169     }
170     if (eq) {
171         copy(fa.begin(), fa.begin() + sz, fb.begin());
172     } else {
173         for (int i = 0; i < (int) b.size(); i++) {
174             int x = (b[i] % m + m) % m;
175             fb[i] = num(x & ((1 << 15) - 1), x >> 15);
176         }
177         fill(fb.begin() + b.size(), fb.begin() + sz, num {0, 0});
178         fft(fb, sz);
179     }
180     dbl ratio = 0.25 / sz;
181     num r2(0, -1);

```

```

182     num r3(ratio, 0);
183     num r4(0, -ratio);
184     num r5(0, 1);
185     for (int i = 0; i <= (sz >> 1); i++) {
186         int j = (sz - i) & (sz - 1);
187         num a1 = (fa[i] + conj(fa[j]));
188         num a2 = (fa[i] - conj(fa[j])) * r2;
189         num b1 = (fb[i] + conj(fb[j])) * r3;
190         num b2 = (fb[i] - conj(fb[j])) * r4;
191         if (i != j) {
192             num c1 = (fa[j] + conj(fa[i]));
193             num c2 = (fa[j] - conj(fa[i])) * r2;
194             num d1 = (fb[j] + conj(fb[i])) * r3;
195             num d2 = (fb[j] - conj(fb[i])) * r4;
196             fa[i] = c1 * d1 + c2 * d2 * r5;
197             fb[i] = c1 * d2 + c2 * d1;
198         }
199         fa[j] = a1 * b1 + a2 * b2 * r5;
200         fb[j] = a1 * b2 + a2 * b1;
201     }
202     fft(fa, sz);
203     fft(fb, sz);
204     vector<int> res(need);
205     for (int i = 0; i < need; i++) {
206         int64_t aa = llround(fa[i].x);
207         int64_t bb = llround(fb[i].x);
208         int64_t cc = llround(fa[i].y);
209         res[i] = static_cast<int>((aa + ((bb % m) << 15) + ((cc % m) << 30)) %
            m);
210     }
211     return res;
212 }
213
214 } // namespace fft
215
216 template <typename T>
217 typename enable_if<is_same<typename Modular<T>::Type, int>::value, vector<
    Modular<T>>>::type operator*(
218     const vector<Modular<T>>& a,
219     const vector<Modular<T>>& b) {
220     if (a.empty() || b.empty()) {
221         return {};

```

```

222     }
223     if (min(a.size(), b.size()) < 150) {
224         vector<Modular<T>> c(a.size() + b.size() - 1, 0);
225         for (int i = 0; i < (int) a.size(); i++) {
226             for (int j = 0; j < (int) b.size(); j++) {
227                 c[i + j] += a[i] * b[j];
228             }
229         }
230         return c;
231     }
232     vector<int> a_mul(a.size());
233     for (int i = 0; i < (int) a.size(); i++) {
234         a_mul[i] = static_cast<int>(a[i]);
235     }
236     vector<int> b_mul(b.size());
237     for (int i = 0; i < (int) b.size(); i++) {
238         b_mul[i] = static_cast<int>(b[i]);
239     }
240     vector<int> c_mul = fft::multiply_mod(a_mul, b_mul, T::value);
241     vector<Modular<T>> c(c_mul.size());
242     for (int i = 0; i < (int) c.size(); i++) {
243         c[i] = c_mul[i];
244     }
245     return c;
246 }
247
248 template <typename T>
249 typename enable_if<is_same<typename Modular<T>::Type, int>::value, vector<
    Modular<T>>>::type& operator*=(
250     vector<Modular<T>>>& a,
251     const vector<Modular<T>>& b) {
252     return a = a * b;
253 }

```

5.5 fwht.cpp

```

1 namespace fwht {
2
3 template<typename T>
4 void hadamard(vector<T> &a) {
5     int n = a.size();

```

```

6     for (int k = 1; k < n; k <= 1) {
7         for (int i = 0; i < n; i += 2 * k) {
8             for (int j = 0; j < k; j++) {
9                 T x = a[i + j];
10                T y = a[i + j + k];
11                a[i + j] = x + y;
12                a[i + j + k] = x - y;
13            }
14        }
15    }
16 }
17
18 template<typename T>
19 vector<T> multiply(vector<T> a, vector<T> b) {
20     int eq = (a == b);
21     int n = 1;
22     while (n < (int) max(a.size(), b.size())) {
23         n <= 1;
24     }
25     a.resize(n);
26     b.resize(n);
27     hadamard(a);
28     if (eq) b = a; else hadamard(b);
29     for (int i = 0; i < n; i++) {
30         a[i] *= b[i];
31     }
32     hadamard(a);
33     T q = 1 / static_cast<T>(n);
34     for (int i = 0; i < n; i++) {
35         a[i] *= q;
36     }
37     return a;
38 }
39
40 } // namespace fwht

```

5.6 gauss.cpp

```

1 const double eps = 1e-9;
2
3 bool IsZero(double v) {

```

```

4   return abs(v) < 1e-9;
5 }
6
7 enum GAUSS_MODE {
8     DEGREE, ABS
9 };
10
11 template <typename T>
12 void GaussianElimination(vector<vector<T>>& a, int limit, GAUSS_MODE mode
    = DEGREE) {
13     if (a.empty() || a[0].empty()) {
14         return;
15     }
16     int h = static_cast<int>(a.size());
17     int w = static_cast<int>(a[0].size());
18     for (int i = 0; i < h; i++) {
19         assert(w == static_cast<int>(a[i].size()));
20     }
21     assert(limit <= w);
22     vector<int> deg(h);
23     for (int i = 0; i < h; i++) {
24         for (int j = 0; j < w; j++) {
25             deg[i] += !IsZero(a[i][j]);
26         }
27     }
28     int r = 0;
29     for (int c = 0; c < limit; c++) {
30         int id = -1;
31         for (int i = r; i < h; i++) {
32             if (!IsZero(a[i][c]) && (id == -1 || (mode == DEGREE && deg[i] < deg
                [id]) || (mode == ABS && abs(a[id][c]) < abs(a[i][c])))) {
33                 id = i;
34             }
35         }
36         if (id == -1) {
37             continue;
38         }
39         if (id > r) {
40             swap(a[r], a[id]);
41             swap(deg[r], deg[id]);
42             for (int j = c; j < w; j++) {
43                 a[id][j] = -a[id][j];

```

```

44         }
45     }
46     vector<int> nonzero;
47     for (int j = c; j < w; j++) {
48         if (!IsZero(a[r][j])) {
49             nonzero.push_back(j);
50         }
51     }
52     T inv_a = 1 / a[r][c];
53     for (int i = r + 1; i < h; i++) {
54         if (IsZero(a[i][c])) {
55             continue;
56         }
57         T coeff = -a[i][c] * inv_a;
58         for (int j : nonzero) {
59             if (!IsZero(a[i][j])) deg[i]--;
60             a[i][j] += coeff * a[r][j];
61             if (!IsZero(a[i][j])) deg[i]++;
62         }
63     }
64     ++r;
65 }
66 for (r = h - 1; r >= 0; r--) {
67     for (int c = 0; c < limit; c++) {
68         if (!IsZero(a[r][c])) {
69             T inv_a = 1 / a[r][c];
70             for (int i = r - 1; i >= 0; i--) {
71                 if (IsZero(a[i][c])) {
72                     continue;
73                 }
74                 T coeff = -a[i][c] * inv_a;
75                 for (int j = c; j < w; j++) {
76                     a[i][j] += coeff * a[r][j];
77                 }
78             }
79             break;
80         }
81     }
82 }
83 }
84
85 template <typename T>

```

```

86 T Determinant(vector<vector<T>> /*@*/ a) {
87     if (a.empty()) {
88         return T{1};
89     }
90     assert(a.size() == a[0].size());
91     GaussianElimination(a, static_cast<int>(a[0].size()));
92     T d{1};
93     for (int i = 0; i < a.h; i++) {
94         d *= a[i][i];
95     }
96     return d;
97 }
98
99 template <typename T>
100 int Rank(vector<vector<T>> /*@*/ a) {
101     if (a.empty()) {
102         return 0;
103     }
104     GaussianElimination(a, static_cast<int>(a[0].size()));
105     int rank = 0;
106     for (int i = 0; i < static_cast<int>(a.size()); i++) {
107         for (int j = 0; j < static_cast<int>(a[i].size()); j++) {
108             if (!IsZero(a[i][j])) {
109                 ++rank;
110                 break;
111             }
112         }
113     }
114     return rank;
115 }
116
117 template <typename T>
118 vector<T> SolveLinearSystem(vector<vector<T>> /*@*/ a, const vector<T>& b,
119                             int w) {
120     int h = static_cast<int>(a.size());
121     assert(h == static_cast<int>(b.size()));
122     if (h > 0) {
123         assert(w == static_cast<int>(a[0].size()));
124     }
125     for (int i = 0; i < h; i++) {
126         a[i].push_back(b[i]);
127     }

```

```

127     GaussianElimination(a, w);
128     vector<T> x(w, 0);
129     for (int i = 0; i < h; i++) {
130         for (int j = 0; j < w; j++) {
131             if (!IsZero(a[i][j])) {
132                 x[j] = a[i][w] / a[i][j];
133                 break;
134             }
135         }
136     }
137     return x;
138 }
139
140 template <typename T>
141 vector<vector<T>> Inverse(vector<vector<T>> /*@*/ a) {
142     if (a.empty()) {
143         return a;
144     }
145     int h = static_cast<int>(a.size());
146     for (int i = 0; i < h; i++) {
147         assert(h == static_cast<int>(a[i].size()));
148     }
149     for (int i = 0; i < h; i++) {
150         a[i].resize(2 * h);
151         a[i][i + h] = 1;
152     }
153     GaussianElimination(a, h);
154     for (int i = 0; i < h; i++) {
155         if (IsZero(a[i][i])) {
156             return {{}};
157         }
158     }
159     vector<vector<T>> b(h);
160     for (int i = 0; i < h; i++) {
161         b[i] = vector<T>(a[i].begin() + h, a[i].end());
162         T coeff = 1 / a[i][i];
163         for (int j = 0; j < h; j++) {
164             b[i][j] *= coeff;
165         }
166     }
167     return b;
168 }

```

5.7 matrix.cpp

```
1 template <typename T>
2 vector<vector<T>> operator*(const vector<vector<T>>& a, const vector<
    vector<T>>& b) {
3     if (a.empty() || b.empty()) {
4         return {{}};
5     }
6     vector<vector<T>> c(a.size(), vector<T>(b[0].size()));
7     for (int i = 0; i < static_cast<int>(c.size()); i++) {
8         for (int j = 0; j < static_cast<int>(c[0].size()); j++) {
9             c[i][j] = 0;
10            for (int k = 0; k < static_cast<int>(b.size()); k++) {
11                c[i][j] += a[i][k] * b[k][j];
12            }
13        }
14    }
15    return c;
16 }
17
18 template <typename T>
19 vector<vector<T>>& operator*=(vector<vector<T>>& a, const vector<vector<T>
    >>& b) {
20     return a = a * b;
21 }
22
23 template <typename T, typename U>
24 vector<vector<T>> power(const vector<vector<T>>& a, const U& b) {
25     assert(b >= 0);
26     vector<U> binary;
27     U bb = b;
28     while (bb > 0) {
29         binary.push_back(bb & 1);
30         bb >>= 1;
31     }
32     vector<vector<T>> res(a.size(), vector<T>(a.size()));
33     for (int i = 0; i < static_cast<int>(a.size()); i++) {
34         res[i][i] = 1;
35     }
36     for (int j = (int) binary.size() - 1; j >= 0; j--) {
37         res *= res;
38         if (binary[j] == 1) {
```

```
39         res *= a;
40     }
41 }
42 return res;
43 }
```

5.8 mint.cpp

```
1 template <typename T>
2 T inverse(T a, T m) {
3     T u = 0, v = 1;
4     while (a != 0) {
5         T t = m / a;
6         m -= t * a; swap(a, m);
7         u -= t * v; swap(u, v);
8     }
9     assert(m == 1);
10    return u;
11 }
12
13 template <typename T>
14 class Modular {
15 public:
16     using Type = typename decay<decltype(T::value)>::type;
17
18     constexpr Modular() : value() {}
19     template <typename U>
20     Modular(const U& x) {
21         value = normalize(x);
22     }
23
24     template <typename U>
25     static Type normalize(const U& x) {
26         Type v;
27         if (-mod() <= x && x < mod()) v = static_cast<Type>(x);
28         else v = static_cast<Type>(x % mod());
29         if (v < 0) v += mod();
30         return v;
31     }
32
33     const Type& operator()() const { return value; }
```



```

34 template <typename U>
35 explicit operator U() const { return static_cast<U>(value); }
36 constexpr static Type mod() { return T::value; }
37
38 Modular& operator+=(const Modular& other) { if ((value += other.value)
    >= mod()) value -= mod(); return *this; }
39 Modular& operator-=(const Modular& other) { if ((value -= other.value) <
    0) value += mod(); return *this; }
40 template <typename U> Modular& operator+=(const U& other) { return *this
    += Modular(other); }
41 template <typename U> Modular& operator-=(const U& other) { return *this
    -= Modular(other); }
42 Modular& operator++() { return *this += 1; }
43 Modular& operator--() { return *this -= 1; }
44 Modular operator++(int) { Modular result(*this); *this += 1; return
    result; }
45 Modular operator--(int) { Modular result(*this); *this -= 1; return
    result; }
46 Modular operator-() const { return Modular(-value); }
47
48 template <typename U = T>
49 typename enable_if<is_same<typename Modular<U>::Type, int>::value,
    Modular>::type& operator==(const Modular& rhs) {
50     value = normalize(static_cast<int64_t>(value) * static_cast<int64_t>(
        rhs.value));
51     return *this;
52 }
53 template <typename U = T>
54 typename enable_if<is_same<typename Modular<U>::Type, long long>::value,
    Modular>::type& operator==(const Modular& rhs) {
55     long long q = static_cast<long long>(static_cast<long double>(value) *
        rhs.value / mod());
56     value = normalize(value * rhs.value - q * mod());
57     return *this;
58 }
59 template <typename U = T>
60 typename enable_if<!is_integral<typename Modular<U>::Type>::value,
    Modular>::type& operator==(const Modular& rhs) {
61     value = normalize(value * rhs.value);
62     return *this;
63 }
64

```

```

65 Modular& operator/=(const Modular& other) { return *this *= Modular(
    inverse(other.value, mod())); }
66
67 friend const Type& abs(const Modular& x) { return x.value; }
68
69 template <typename U>
70 friend bool operator==(const Modular<U>& lhs, const Modular<U>& rhs);
71
72 template <typename U>
73 friend bool operator<(const Modular<U>& lhs, const Modular<U>& rhs);
74
75 template <typename V, typename U>
76 friend V& operator>>(V& stream, Modular<U>& number);
77
78 private:
79     Type value;
80 };
81
82 template <typename T> bool operator==(const Modular<T>& lhs, const Modular
    <T>& rhs) { return lhs.value == rhs.value; }
83 template <typename T, typename U> bool operator==(const Modular<T>& lhs, U
    rhs) { return lhs == Modular<T>(rhs); }
84 template <typename T, typename U> bool operator==(U lhs, const Modular<T>&
    rhs) { return Modular<T>(lhs) == rhs; }
85
86 template <typename T> bool operator!=(const Modular<T>& lhs, const Modular
    <T>& rhs) { return !(lhs == rhs); }
87 template <typename T, typename U> bool operator!=(const Modular<T>& lhs, U
    rhs) { return !(lhs == rhs); }
88 template <typename T, typename U> bool operator!=(U lhs, const Modular<T>&
    rhs) { return !(lhs == rhs); }
89
90 template <typename T> bool operator<(const Modular<T>& lhs, const Modular<
    T>& rhs) { return lhs.value < rhs.value; }
91
92 template <typename T> Modular<T> operator+(const Modular<T>& lhs, const
    Modular<T>& rhs) { return Modular<T>(lhs) += rhs; }
93 template <typename T, typename U> Modular<T> operator+(const Modular<T>&
    lhs, U rhs) { return Modular<T>(lhs) += rhs; }
94 template <typename T, typename U> Modular<T> operator+(U lhs, const
    Modular<T>& rhs) { return Modular<T>(lhs) += rhs; }
95

```

```

96 template <typename T> Modular<T> operator-(const Modular<T>& lhs, const
    Modular<T>& rhs) { return Modular<T>(lhs) -= rhs; }
97 template <typename T, typename U> Modular<T> operator-(const Modular<T>&
    lhs, U rhs) { return Modular<T>(lhs) -= rhs; }
98 template <typename T, typename U> Modular<T> operator-(U lhs, const
    Modular<T>& rhs) { return Modular<T>(lhs) -= rhs; }
99
100 template <typename T> Modular<T> operator*(const Modular<T>& lhs, const
    Modular<T>& rhs) { return Modular<T>(lhs) *= rhs; }
101 template <typename T, typename U> Modular<T> operator*(const Modular<T>&
    lhs, U rhs) { return Modular<T>(lhs) *= rhs; }
102 template <typename T, typename U> Modular<T> operator*(U lhs, const
    Modular<T>& rhs) { return Modular<T>(lhs) *= rhs; }
103
104 template <typename T> Modular<T> operator/(const Modular<T>& lhs, const
    Modular<T>& rhs) { return Modular<T>(lhs) /= rhs; }
105 template <typename T, typename U> Modular<T> operator/(const Modular<T>&
    lhs, U rhs) { return Modular<T>(lhs) /= rhs; }
106 template <typename T, typename U> Modular<T> operator/(U lhs, const
    Modular<T>& rhs) { return Modular<T>(lhs) /= rhs; }
107
108 template<typename T, typename U>
109 Modular<T> power(const Modular<T>& a, const U& b) {
110     assert(b >= 0);
111     Modular<T> x = a, res = 1;
112     U p = b;
113     while (p > 0) {
114         if (p & 1) res *= x;
115         x *= x;
116         p >>= 1;
117     }
118     return res;
119 }
120
121 template <typename T>
122 bool IsZero(const Modular<T>& number) {
123     return number() == 0;
124 }
125
126 template <typename T>
127 string to_string(const Modular<T>& number) {
128     return to_string(number());

```

```

129 }
130
131 // U == std::ostream? but done this way because of fastoutput
132 template <typename U, typename T>
133 U& operator<<(U& stream, const Modular<T>& number) {
134     return stream << number();
135 }
136
137 // U == std::istream? but done this way because of fastinput
138 template <typename U, typename T>
139 U& operator>>(U& stream, Modular<T>& number) {
140     typename common_type<typename Modular<T>::Type, long long>::type x;
141     stream >> x;
142     number.value = Modular<T>::normalize(x);
143     return stream;
144 }
145
146 /*
147 using ModType = int;
148
149 struct VarMod { static ModType value; };
150 ModType VarMod::value;
151 ModType& md = VarMod::value;
152 using Mint = Modular<VarMod>;
153 */
154
155 constexpr int md = 0;
156 using Mint = Modular<std::integral_constant<decay<decltype(md)>::type, md
    >>;
157
158 /*vector<Mint> fact(1, 1);
159 vector<Mint> inv_fact(1, 1);
160
161 Mint C(int n, int k) {
162     if (k < 0 || k > n) {
163         return 0;
164     }
165     while ((int) fact.size() < n + 1) {
166         fact.push_back(fact.back() * (int) fact.size());
167         inv_fact.push_back(1 / fact.back());
168     }
169     return fact[n] * inv_fact[k] * inv_fact[n - k];

```

```
170 }*/
```

5.9 ntt.cpp

```
1  template <typename T>
2  class NTT {
3  public:
4      using Type = typename decay<decltype(T::value)>::type;
5
6      static Type md;
7      static Modular<T> root;
8      static int base;
9      static int max_base;
10     static vector<Modular<T>> roots;
11     static vector<int> rev;
12
13     static void clear() {
14         root = 0;
15         base = 0;
16         max_base = 0;
17         roots.clear();
18         rev.clear();
19     }
20
21     static void init() {
22         md = T::value;
23         assert(md >= 3 && md % 2 == 1);
24         auto tmp = md - 1;
25         max_base = 0;
26         while (tmp % 2 == 0) {
27             tmp /= 2;
28             max_base++;
29         }
30         root = 2;
31         while (power(root, (md - 1) >> 1) == 1) {
32             root++;
33         }
34         assert(power(root, md - 1) == 1);
35         root = power(root, (md - 1) >> max_base);
36         base = 1;
37         rev = {0, 1};
```

```
38     roots = {0, 1};
39 }
40
41 static void ensure_base(int nbase) {
42     if (md != T::value) {
43         clear();
44     }
45     if (roots.empty()) {
46         init();
47     }
48     if (nbase <= base) {
49         return;
50     }
51     assert(nbase <= max_base);
52     rev.resize(1 << nbase);
53     for (int i = 0; i < (1 << nbase); i++) {
54         rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (nbase - 1));
55     }
56     roots.resize(1 << nbase);
57     while (base < nbase) {
58         Modular<T> z = power(root, 1 << (max_base - 1 - base));
59         for (int i = 1 << (base - 1); i < (1 << base); i++) {
60             roots[i << 1] = roots[i];
61             roots[(i << 1) + 1] = roots[i] * z;
62         }
63         base++;
64     }
65 }
66
67 static void fft(vector<Modular<T>> &a) {
68     int n = (int) a.size();
69     assert((n & (n - 1)) == 0);
70     int zeros = __builtin_ctz(n);
71     ensure_base(zeros);
72     int shift = base - zeros;
73     for (int i = 0; i < n; i++) {
74         if (i < (rev[i] >> shift)) {
75             swap(a[i], a[rev[i] >> shift]);
76         }
77     }
78     for (int k = 1; k < n; k <= 1) {
79         for (int i = 0; i < n; i += 2 * k) {
```

```

80     for (int j = 0; j < k; j++) {
81         Modular<T> x = a[i + j];
82         Modular<T> y = a[i + j + k] * roots[j + k];
83         a[i + j] = x + y;
84         a[i + j + k] = x - y;
85     }
86 }
87 }
88 }
89
90 static vector<Modular<T>> multiply(vector<Modular<T>> a, vector<Modular<
    T>> b) {
91     if (a.empty() || b.empty()) {
92         return {};
93     }
94     int eq = (a == b);
95     int need = (int) a.size() + (int) b.size() - 1;
96     int nbase = 0;
97     while ((1 << nbase) < need) nbase++;
98     ensure_base(nbase);
99     int sz = 1 << nbase;
100    a.resize(sz);
101    b.resize(sz);
102    fft(a);
103    if (eq) b = a; else fft(b);
104    Modular<T> inv_sz = 1 / static_cast<Modular<T>>(sz);
105    for (int i = 0; i < sz; i++) {
106        a[i] *= b[i] * inv_sz;
107    }
108    reverse(a.begin() + 1, a.end());
109    fft(a);
110    a.resize(need);
111    return a;
112 }
113 };
114
115 template <typename T> typename NTT<T>::Type NTT<T>::md;
116 template <typename T> Modular<T> NTT<T>::root;
117 template <typename T> int NTT<T>::base;
118 template <typename T> int NTT<T>::max_base;
119 template <typename T> vector<Modular<T>> NTT<T>::roots;
120 template <typename T> vector<int> NTT<T>::rev;

```

```

121
122 template <typename T>
123 vector<Modular<T>> inverse(const vector<Modular<T>>& a) {
124     assert(!a.empty());
125     int n = (int) a.size();
126     vector<Modular<T>> b = {1 / a[0]};
127     while ((int) b.size() < n) {
128         vector<Modular<T>> x(a.begin(), a.begin() + min(a.size(), b.size() <<
            1));
129         x.resize(b.size() << 1);
130         b.resize(b.size() << 1);
131         vector<Modular<T>> c = b;
132         NTT<T>::fft(c);
133         NTT<T>::fft(x);
134         Modular<T> inv = 1 / static_cast<Modular<T>>((int) x.size());
135         for (int i = 0; i < (int) x.size(); i++) {
136             x[i] *= c[i] * inv;
137         }
138         reverse(x.begin() + 1, x.end());
139         NTT<T>::fft(x);
140         rotate(x.begin(), x.begin() + (x.size() >> 1), x.end());
141         fill(x.begin() + (x.size() >> 1), x.end(), 0);
142         NTT<T>::fft(x);
143         for (int i = 0; i < (int) x.size(); i++) {
144             x[i] *= c[i] * inv;
145         }
146         reverse(x.begin() + 1, x.end());
147         NTT<T>::fft(x);
148         for (int i = 0; i < ((int) x.size() >> 1); i++) {
149             b[i + ((int) x.size() >> 1)] = -x[i];
150         }
151     }
152     b.resize(n);
153     return b;
154 }
155
156 template <typename T>
157 vector<Modular<T>> inverse_old(vector<Modular<T>> a) {
158     assert(!a.empty());
159     int n = (int) a.size();
160     if (n == 1) {
161         return {1 / a[0]};

```

```

162     }
163     int m = (n + 1) >> 1;
164     vector<Modular<T>> b = inverse(vector<Modular<T>>(a.begin(), a.begin() +
165         m));
166     int need = n << 1;
167     int nbase = 0;
168     while ((1 << nbase) < need) {
169         ++nbase;
170     }
171     NTT<T>::ensure_base(nbase);
172     int size = 1 << nbase;
173     a.resize(size);
174     b.resize(size);
175     NTT<T>::fft(a);
176     NTT<T>::fft(b);
177     Modular<T> inv = 1 / static_cast<Modular<T>>(size);
178     for (int i = 0; i < size; ++i) {
179         a[i] = (2 - a[i] * b[i]) * b[i] * inv;
180     }
181     reverse(a.begin() + 1, a.end());
182     NTT<T>::fft(a);
183     a.resize(n);
184     return a;
185 }
186 template <typename T>
187 vector<Modular<T>> operator*(const vector<Modular<T>>& a, const vector<
188     Modular<T>>& b) {
189     if (a.empty() || b.empty()) {
190         return {};
191     }
192     if (min(a.size(), b.size()) < 150) {
193         vector<Modular<T>> c(a.size() + b.size() - 1, 0);
194         for (int i = 0; i < (int) a.size(); i++) {
195             for (int j = 0; j < (int) b.size(); j++) {
196                 c[i + j] += a[i] * b[j];
197             }
198         }
199         return c;
200     }
201     return NTT<T>::multiply(a, b);
202 }

```

```

202
203 template <typename T>
204 vector<Modular<T>>& operator*=(vector<Modular<T>>& a, const vector<Modular
205     <T>>& b) {
206     return a = a * b;
207 }

```

5.10 poly.cpp

```

1 template <typename T>
2 vector<T>& operator+=(vector<T>& a, const vector<T>& b) {
3     if (a.size() < b.size()) {
4         a.resize(b.size());
5     }
6     for (int i = 0; i < (int) b.size(); i++) {
7         a[i] += b[i];
8     }
9     return a;
10 }
11
12 template <typename T>
13 vector<T> operator+(const vector<T>& a, const vector<T>& b) {
14     vector<T> c = a;
15     return c += b;
16 }
17
18 template <typename T>
19 vector<T>& operator-=(vector<T>& a, const vector<T>& b) {
20     if (a.size() < b.size()) {
21         a.resize(b.size());
22     }
23     for (int i = 0; i < (int) b.size(); i++) {
24         a[i] -= b[i];
25     }
26     return a;
27 }
28
29 template <typename T>
30 vector<T> operator-(const vector<T>& a, const vector<T>& b) {
31     vector<T> c = a;
32     return c -= b;
33 }

```

```

33 }
34
35 template <typename T>
36 vector<T> operator-(const vector<T>& a) {
37     vector<T> c = a;
38     for (int i = 0; i < (int) c.size(); i++) {
39         c[i] = -c[i];
40     }
41     return c;
42 }
43
44 template <typename T>
45 vector<T> operator*(const vector<T>& a, const vector<T>& b) {
46     if (a.empty() || b.empty()) {
47         return {};
48     }
49     vector<T> c(a.size() + b.size() - 1, 0);
50     for (int i = 0; i < (int) a.size(); i++) {
51         for (int j = 0; j < (int) b.size(); j++) {
52             c[i + j] += a[i] * b[j];
53         }
54     }
55     return c;
56 }
57
58 template <typename T>
59 vector<T>& operator*=(vector<T>& a, const vector<T>& b) {
60     return a = a * b;
61 }
62
63 template <typename T>
64 vector<T> inverse(const vector<T>& a) {
65     assert(!a.empty());
66     int n = (int) a.size();
67     vector<T> b = {1 / a[0]};
68     while ((int) b.size() < n) {
69         vector<T> a_cut(a.begin(), a.begin() + min(a.size(), b.size() << 1));
70         vector<T> x = b * b * a_cut;
71         b.resize(b.size() << 1);
72         for (int i = (int) b.size() >> 1; i < (int) min(x.size(), b.size()); i
            ++){
73             b[i] = -x[i];

```

```

74     }
75 }
76 b.resize(n);
77 return b;
78 }
79
80 template <typename T>
81 vector<T>& operator/=(vector<T>& a, const vector<T>& b) {
82     int n = (int) a.size();
83     int m = (int) b.size();
84     if (n < m) {
85         a.clear();
86     } else {
87         vector<T> d = b;
88         reverse(a.begin(), a.end());
89         reverse(d.begin(), d.end());
90         d.resize(n - m + 1);
91         a *= inverse(d);
92         a.erase(a.begin() + n - m + 1, a.end());
93         reverse(a.begin(), a.end());
94     }
95     return a;
96 }
97
98 template <typename T>
99 vector<T> operator/(const vector<T>& a, const vector<T>& b) {
100     vector<T> c = a;
101     return c /= b;
102 }
103
104 template <typename T>
105 vector<T>& operator%=(vector<T>& a, const vector<T>& b) {
106     int n = (int) a.size();
107     int m = (int) b.size();
108     if (n >= m) {
109         vector<T> c = (a / b) * b;
110         a.resize(m - 1);
111         for (int i = 0; i < m - 1; i++) {
112             a[i] -= c[i];
113         }
114     }
115     return a;

```

```

116 }
117
118 template <typename T>
119 vector<T> operator%(const vector<T>& a, const vector<T>& b) {
120     vector<T> c = a;
121     return c %= b;
122 }
123
124 template <typename T, typename U>
125 vector<T> power(const vector<T>& a, const U& b, const vector<T>& c) {
126     assert(b >= 0);
127     vector<U> binary;
128     U bb = b;
129     while (bb > 0) {
130         binary.push_back(bb & 1);
131         bb >>= 1;
132     }
133     vector<T> res = vector<T>{1} % c;
134     for (int j = (int) binary.size() - 1; j >= 0; j--) {
135         res = res * res % c;
136         if (binary[j] == 1) {
137             res = res * a % c;
138         }
139     }
140     return res;
141 }
142
143 template <typename T>
144 vector<T> derivative(const vector<T>& a) {
145     vector<T> c = a;
146     for (int i = 0; i < (int) c.size(); i++) {
147         c[i] *= i;
148     }
149     if (!c.empty()) {
150         c.erase(c.begin());
151     }
152     return c;
153 }
154
155 template <typename T>
156 vector<T> primitive(const vector<T>& a) {
157     vector<T> c = a;

```

```

158     c.insert(c.begin(), 0);
159     for (int i = 1; i < (int) c.size(); i++) {
160         c[i] /= i;
161     }
162     return c;
163 }
164
165 template <typename T>
166 vector<T> logarithm(const vector<T>& a) {
167     assert(!a.empty() && a[0] == 1);
168     vector<T> u = primitive(derivative(a) * inverse(a));
169     u.resize(a.size());
170     return u;
171 }
172
173 template <typename T>
174 vector<T> exponent(const vector<T>& a) {
175     assert(!a.empty() && a[0] == 0);
176     int n = (int) a.size();
177     vector<T> b = {1};
178     while ((int) b.size() < n) {
179         vector<T> x(a.begin(), a.begin() + min(a.size(), b.size() << 1));
180         x[0] += 1;
181         vector<T> old_b = b;
182         b.resize(b.size() << 1);
183         x -= logarithm(b);
184         x *= old_b;
185         for (int i = (int) b.size() >> 1; i < (int) min(x.size(), b.size()); i
            ++){
186             b[i] = x[i];
187         }
188     }
189     b.resize(n);
190     return b;
191 }
192
193 template <typename T>
194 vector<T> sqrt(const vector<T>& a) {
195     assert(!a.empty() && a[0] == 1);
196     int n = (int) a.size();
197     vector<T> b = {1};
198     while ((int) b.size() < n) {

```

```

199     vector<T> x(a.begin(), a.begin() + min(a.size(), b.size() << 1));
200     b.resize(b.size() << 1);
201     x *= inverse(b);
202     T inv2 = 1 / static_cast<T>(2);
203     for (int i = (int) b.size() >> 1; i < (int) min(x.size(), b.size()); i
        ++ ) {
204         b[i] = x[i] * inv2;
205     }
206 }
207 b.resize(n);
208 return b;
209 }
210
211 template <typename T>
212 vector<T> multiply(const vector<vector<T>>& a) {
213     if (a.empty()) {
214         return {0};
215     }
216     function<vector<T>(int, int)> mult = [&](int l, int r) {
217         if (l == r) {
218             return a[l];
219         }
220         int y = (l + r) >> 1;
221         return mult(l, y) * mult(y + 1, r);
222     };
223     return mult(0, (int) a.size() - 1);
224 }
225
226 template <typename T>
227 T evaluate(const vector<T>& a, const T& x) {
228     T res = 0;
229     for (int i = (int) a.size() - 1; i >= 0; i--) {
230         res = res * x + a[i];
231     }
232     return res;
233 }
234
235 template <typename T>
236 vector<T> evaluate(const vector<T>& a, const vector<T>& x) {
237     if (x.empty()) {
238         return {};
239     }

```

```

240     if (a.empty()) {
241         return vector<T>(x.size(), 0);
242     }
243     int n = (int) x.size();
244     vector<vector<T>> st((n << 1) - 1);
245     function<void(int, int, int)> build = [&](int v, int l, int r) {
246         if (l == r) {
247             st[v] = vector<T>{-x[l], 1};
248         } else {
249             int y = (l + r) >> 1;
250             int z = v + ((y - l + 1) << 1);
251             build(v + 1, l, y);
252             build(z, y + 1, r);
253             st[v] = st[v + 1] * st[z];
254         }
255     };
256     build(0, 0, n - 1);
257     vector<T> res(n);
258     function<void(int, int, int, vector<T>>> eval = [&](int v, int l, int r,
        vector<T> f) {
259         f %= st[v];
260         if ((int) f.size() < 150) {
261             for (int i = l; i <= r; i++) {
262                 res[i] = evaluate(f, x[i]);
263             }
264             return;
265         }
266         if (l == r) {
267             res[l] = f[0];
268         } else {
269             int y = (l + r) >> 1;
270             int z = v + ((y - l + 1) << 1);
271             eval(v + 1, l, y, f);
272             eval(z, y + 1, r, f);
273         }
274     };
275     eval(0, 0, n - 1, a);
276     return res;
277 }
278
279 template <typename T>
280 vector<T> interpolate(const vector<T>& x, const vector<T>& y) {

```



```

281     if (x.empty()) {
282         return {};
283     }
284     assert(x.size() == y.size());
285     int n = (int) x.size();
286     vector<vector<T>> st((n << 1) - 1);
287     function<void(int, int, int)>> build = [&](int v, int l, int r) {
288         if (l == r) {
289             st[v] = vector<T>{-x[l], 1};
290         } else {
291             int w = (l + r) >> 1;
292             int z = v + ((w - l + 1) << 1);
293             build(v + 1, l, w);
294             build(z, w + 1, r);
295             st[v] = st[v + 1] * st[z];
296         }
297     };
298     build(0, 0, n - 1);
299     vector<T> m = st[0];
300     vector<T> dm = derivative(m);
301     vector<T> val(n);
302     function<void(int, int, int, vector<T>>> eval = [&](int v, int l, int r,
303         vector<T> f) {
304         f %= st[v];
305         if ((int) f.size() < 150) {
306             for (int i = l; i <= r; i++) {
307                 val[i] = evaluate(f, x[i]);
308             }
309             return;
310         }
311         if (l == r) {
312             val[l] = f[0];
313         } else {
314             int w = (l + r) >> 1;
315             int z = v + ((w - l + 1) << 1);
316             eval(v + 1, l, w, f);
317             eval(z, w + 1, r, f);
318         }
319     };
320     eval(0, 0, n - 1, dm);
321     for (int i = 0; i < n; i++) {
322         val[i] = y[i] / val[i];

```

```

322     }
323     function<vector<T>(int, int, int)>> calc = [&](int v, int l, int r) {
324         if (l == r) {
325             return vector<T>{val[l]};
326         }
327         int w = (l + r) >> 1;
328         int z = v + ((w - l + 1) << 1);
329         return calc(v + 1, l, w) * st[z] + calc(z, w + 1, r) * st[v + 1];
330     };
331     return calc(0, 0, n - 1);
332 }
333
334 //  $f[i] = 1^i + 2^i + \dots + up^i$ 
335 template <typename T>
336 vector<T> faulhaber(const T& up, int n) {
337     vector<T> ex(n + 1);
338     T e = 1;
339     for (int i = 0; i <= n; i++) {
340         ex[i] = e;
341         e /= i + 1;
342     }
343     vector<T> den = ex;
344     den.erase(den.begin());
345     for (auto& d : den) {
346         d = -d;
347     }
348     vector<T> num(n);
349     T p = 1;
350     for (int i = 0; i < n; i++) {
351         p *= up + 1;
352         num[i] = ex[i + 1] * (1 - p);
353     }
354     vector<T> res = num * inverse(den);
355     res.resize(n);
356     T f = 1;
357     for (int i = 0; i < n; i++) {
358         res[i] *= f;
359         f *= i + 1;
360     }
361     return res;
362 }
363

```

```

364 // (x + 1) * (x + 2) * ... * (x + n)
365 // (can be optimized with precomputed inverses)
366 template <typename T>
367 vector<T> sequence(int n) {
368     if (n == 0) {
369         return {1};
370     }
371     if (n % 2 == 1) {
372         return sequence<T>(n - 1) * vector<T>{n, 1};
373     }
374     vector<T> c = sequence<T>(n / 2);
375     vector<T> a = c;
376     reverse(a.begin(), a.end());
377     T f = 1;
378     for (int i = n / 2 - 1; i >= 0; i--) {
379         f *= n / 2 - i;
380         a[i] *= f;
381     }
382     vector<T> b(n / 2 + 1);
383     b[0] = 1;
384     for (int i = 1; i <= n / 2; i++) {
385         b[i] = b[i - 1] * (n / 2) / i;
386     }
387     vector<T> h = a * b;
388     h.resize(n / 2 + 1);
389     reverse(h.begin(), h.end());
390     f = 1;
391     for (int i = 1; i <= n / 2; i++) {
392         f /= i;
393         h[i] *= f;
394     }
395     vector<T> res = c * h;
396     return res;
397 }
398
399 template <typename T>
400 class OnlineProduct {
401 public:
402     const vector<T> a;
403     vector<T> b;
404     vector<T> c;
405

```

```

406     OnlineProduct(const vector<T>& a_) : a(a_) {}
407
408     T add(const T& val) {
409         int i = (int) b.size();
410         b.push_back(val);
411         if ((int) c.size() <= i) {
412             c.resize(i + 1);
413         }
414         c[i] += a[0] * b[i];
415         int z = 1;
416         while ((i & (z - 1)) == z - 1 && (int) a.size() > z) {
417             vector<T> a_mul(a.begin() + z, a.begin() + min(z << 1, (int) a.size
418                 ()));
419             vector<T> b_mul(b.end() - z, b.end());
420             vector<T> c_mul = a_mul * b_mul;
421             if ((int) c.size() <= i + (int) c_mul.size()) {
422                 c.resize(i + c_mul.size() + 1);
423             }
424             for (int j = 0; j < (int) c_mul.size(); j++) {
425                 c[i + 1 + j] += c_mul[j];
426             }
427             z <<= 1;
428         }
429         return c[i];
430     };

```

5.11 primitive.cpp

```

1  template <typename T>
2  struct PrimitiveVarMod { static T value; };
3  template <typename T>
4  T PrimitiveVarMod<T>::value;
5
6  template <typename T, class F>
7  T GetPrimitiveRoot(const T& modulo, const F& factorize) {
8      if (modulo <= 0) {
9          return -1;
10     }
11     if (modulo == 1 || modulo == 2 || modulo == 4) {
12         return modulo - 1;

```

```

13     }
14     vector<pair<T, int>> modulo_factors = factorize(modulo);
15     if (modulo_factors[0].first == 2 && (modulo_factors[0].second != 1 ||
        modulo_factors.size() != 2)) {
16         return -1;
17     }
18     if (modulo_factors[0].first != 2 && modulo_factors.size() != 1) {
19         return -1;
20     }
21     set<T> phi_factors;
22     T phi = modulo;
23     for (auto& d : modulo_factors) {
24         phi = phi / d.first * (d.first - 1);
25         if (d.second > 1) {
26             phi_factors.insert(d.first);
27         }
28         for (auto& e : factorize(d.first - 1)) {
29             phi_factors.insert(e.first);
30         }
31     }
32     PrimitiveVarMod<T>::value = modulo;
33     Modular<PrimitiveVarMod<T>> gen = 2;
34     while (gen != 0) {
35         if (power(gen, phi) != 1) {
36             continue;
37         }
38         bool ok = true;
39         for (auto& p : phi_factors) {
40             if (power(gen, phi / p) == 1) {
41                 ok = false;
42                 break;
43             }
44         }
45         if (ok) {
46             return gen();
47         }
48         gen++;
49     }
50     assert(false);
51     return -1;
52 }
53

```

```

54 template <typename T>
55 T GetPrimitiveRoot(const T& modulo) {
56     return GetPrimitiveRoot(modulo, factorizer::Factorize<T>());
57 }

```

5.12 simplex.cpp

```

1  typedef long double ld;
2
3  const ld eps = 1e-8;
4
5  vector<ld> simplex(vector<vector<ld>> a) {
6      int n = (int) a.size() - 1;
7      int m = (int) a[0].size() - 1;
8      vector<int> left(n + 1);
9      vector<int> up(m + 1);
10     iota(left.begin(), left.end(), m);
11     iota(up.begin(), up.end(), 0);
12     auto pivot = [&](int x, int y) {
13         swap(left[x], up[y]);
14         ld k = a[x][y];
15         a[x][y] = 1;
16         vector<int> pos;
17         for (int j = 0; j <= m; j++) {
18             a[x][j] /= k;
19             if (fabs(a[x][j]) > eps) {
20                 pos.push_back(j);
21             }
22         }
23         for (int i = 0; i <= n; i++) {
24             if (fabs(a[i][y]) < eps || i == x) {
25                 continue;
26             }
27             k = a[i][y];
28             a[i][y] = 0;
29             for (int j : pos) {
30                 a[i][j] -= k * a[x][j];
31             }
32         }
33     };
34     while (1) {

```

```

35     int x = -1;
36     for (int i = 1; i <= n; i++) {
37         if (a[i][0] < -eps && (x == -1 || a[i][0] < a[x][0])) {
38             x = i;
39         }
40     }
41     if (x == -1) {
42         break;
43     }
44     int y = -1;
45     for (int j = 1; j <= m; j++) {
46         if (a[x][j] < -eps && (y == -1 || a[x][j] < a[x][y])) {
47             y = j;
48         }
49     }
50     if (y == -1) {
51         return vector<ld>(); // infeasible
52     }
53     pivot(x, y);
54 }
55 while (1) {
56     int y = -1;
57     for (int j = 1; j <= m; j++) {
58         if (a[0][j] > eps && (y == -1 || a[0][j] > a[0][y])) {
59             y = j;
60         }
61     }
62     if (y == -1) {
63         break;
64     }
65     int x = -1;
66     for (int i = 1; i <= n; i++) {
67         if (a[i][y] > eps && (x == -1 || a[i][0] / a[i][y] < a[x][0] / a[x][
68             y])) {
69             x = i;
70         }
71     }
72     if (x == -1) {
73         return vector<ld>(); // unbounded
74     }
75     pivot(x, y);

```

```

76     vector<ld> ans(m + 1);
77     for (int i = 1; i <= n; i++) {
78         if (left[i] <= m) {
79             ans[left[i]] = a[i][0];
80         }
81     }
82     ans[0] = -a[0][0];
83     return ans;
84 }

```

5.13 sparsematrix.cpp

```

1  const double eps = 1e-9;
2
3  bool IsZero(double v) {
4      return abs(v) < 1e-9;
5  }
6
7  template <typename T>
8  class SparseMatrix {
9      public:
10     int h;
11     int w;
12     vector<map<int, T>> rows;
13     vector<map<int, T>> cols;
14
15     SparseMatrix(int h_, int w_) : h(h_), w(w_) {
16         rows.resize(h);
17         cols.resize(w);
18     }
19
20     void set(int i, int j, const T& value) {
21         if (IsZero(value)) {
22             rows[i].erase(j);
23             cols[j].erase(i);
24         } else {
25             rows[i][j] = value;
26             cols[j][i] = value;
27         }
28     }
29

```

```

30 void modify(int i, int j, const T& value) {
31     if (IsZero(value)) {
32         return;
33     }
34     auto it = rows[i].find(j);
35     if (it == rows[i].end()) {
36         rows[i][j] = value;
37         cols[j][i] = value;
38     } else {
39         it->second += value;
40         if (IsZero(it->second)) {
41             rows[i].erase(it);
42             cols[j].erase(i);
43         } else {
44             cols[j][i] = it->second;
45         }
46     }
47 }
48
49 T get(int i, int j) {
50     auto it = rows[i].find(j);
51     if (it == rows[i].end()) {
52         return T{};
53     }
54     return it->second;
55 }
56
57 void transpose() {
58     swap(h, w);
59     swap(rows, cols);
60 }
61 };
62
63 template <typename T>
64 void GaussianElimination(SparseMatrix<T>& a, int limit) {
65     assert(limit <= a.w);
66     int r = 0;
67     for (int c = 0; c < limit; c++) {
68         int mn = a.w + 1;
69         int id = -1;
70         for (auto& p : a.cols[c]) {
71             int i = p.first;

```

```

72             if (i >= r) {
73                 int sz = static_cast<int>(a.rows[i].size());
74                 if (sz < mn) {
75                     mn = sz;
76                     id = i;
77                 }
78             }
79         }
80         if (id == -1) {
81             continue;
82         }
83         if (id > r) {
84             set<int> s;
85             for (auto& p : a.rows[r]) {
86                 s.insert(p.first);
87             }
88             for (auto& p : a.rows[id]) {
89                 s.insert(p.first);
90             }
91             for (int j : s) {
92                 T tmp = a.get(r, j);
93                 a.set(r, j, a.get(id, j));
94                 a.set(id, j, -tmp);
95             }
96         }
97         T inv_a = 1 / a.get(r, c);
98         vector<int> touched_rows;
99         for (auto& p : a.cols[c]) {
100             int i = p.first;
101             if (i > r) {
102                 touched_rows.push_back(i);
103                 T coeff = -p.second * inv_a;
104                 for (auto& q : a.rows[r]) {
105                     if (q.first != c) {
106                         a.modify(i, q.first, coeff * q.second);
107                     }
108                 }
109             }
110         }
111         for (int i : touched_rows) {
112             a.set(i, c, 0);
113         }

```

```

114     ++r;
115 }
116 }
117
118 template <typename T>
119 T Determinant(SparseMatrix<T> /*&a*/ a) {
120     assert(a.h == a.w);
121     GaussianElimination(a, a.w);
122     T d{1};
123     for (int i = 0; i < a.h; i++) {
124         d *= a.get(i, i);
125     }
126     return d;
127 }
128
129 template <typename T>
130 int Rank(SparseMatrix<T> /*&a*/ a) {
131     GaussianElimination(a, a.w);
132     int rank = 0;
133     for (int i = 0; i < a.h; i++) {
134         if (!a.rows[i].empty()) {
135             ++rank;
136         }
137     }
138     return rank;
139 }
140
141 template <typename T>
142 vector<T> SolveLinearSystem(SparseMatrix<T> /*&a*/ a, const vector<T>& b) {
143     assert(a.h == static_cast<int>(b.size()));
144     ++a.w;
145     a.cols.emplace_back();
146     for (int i = 0; i < a.h; i++) {
147         a.set(i, a.w - 1, b[i]);
148     }
149     GaussianElimination(a, a.w - 1);
150     vector<T> x(a.h, 0);
151     for (int r = a.h - 1; r >= 0; r--) {
152         int c = a.rows[r].begin()->first;
153         if (c == a.w - 1) {
154             return {};
155         }

```

```

156         x[c] = a.get(r, a.w - 1) / a.get(r, c);
157         vector<int> touched_rows;
158         for (auto& q : a.cols[c]) {
159             int i = q.first;
160             if (i < r) {
161                 touched_rows.push_back(i);
162                 a.modify(i, a.w - 1, -x[c] * q.second);
163             }
164         }
165         for (int i : touched_rows) {
166             a.set(i, c, 0);
167         }
168     }
169     return x;
170 }

```

6 string

6.1 duval.cpp

```

1 template <typename T>
2 int duval(int n, const T &s) {
3     assert(n >= 1);
4     int i = 0, ans = 0;
5     while (i < n) {
6         ans = i;
7         int j = i + 1, k = i;
8         while (j < n + n && !(s[j % n] < s[k % n])) {
9             if (s[k % n] < s[j % n]) {
10                 k = i;
11             } else {
12                 k++;
13             }
14             j++;
15         }
16         while (i <= k) {
17             i += j - k;
18         }
19     }
20     return ans;
21     // returns 0-indexed position of the least cyclic shift

```

```

22 }
23
24 template <typename T>
25 int duval(const T &s) {
26     return duval((int) s.size(), s);
27 }

```

6.2 duval-prefixes.cpp

```

1  template <typename T>
2  vector<int> duval_prefixes(int n, const T &s) {
3      vector<int> z = z_function(n, s);
4      vector<int> ans(n, 0);
5      int i = 0, pos = 0;
6      while (i < n) {
7          int j = i, k = i;
8          while (j < n) {
9              j++;
10             if (j > pos) {
11                 if (z[k] <= pos - k && s[z[k]] < s[k + z[k]]) {
12                     int shift = (pos - i) / (j - k) * (j - k);
13                     ans[pos] = ans[pos - shift] + shift;
14                 } else {
15                     ans[pos] = i;
16                 }
17                 pos++;
18             }
19             if (s[k] < s[j]) k = i; else
20             if (!(s[j] < s[k])) k++; else
21             else break;
22         }
23         while (i <= k) {
24             i += j - k;
25         }
26     }
27     return ans;
28     // returns 0-indexed positions of the least cyclic shifts of all
       prefixes
29 }
30
31 template <typename T>

```

```

32 vector<int> duval_prefixes(const T &s) {
33     return duval_prefixes((int) s.size(), s);
34 }

```

6.3 hash61.cpp

```

1  struct hash61 {
2      static const uint64_t md = (1LL << 61) - 1;
3      static uint64_t step;
4      static vector<uint64_t> pw;
5
6      uint64_t addmod(uint64_t a, uint64_t b) const {
7          a += b;
8          if (a >= md) a -= md;
9          return a;
10     }
11
12     uint64_t submod(uint64_t a, uint64_t b) const {
13         a += md - b;
14         if (a >= md) a -= md;
15         return a;
16     }
17
18     uint64_t mulmod(uint64_t a, uint64_t b) const {
19         uint64_t l1 = (uint32_t) a, h1 = a >> 32, l2 = (uint32_t) b, h2 = b >>
20             32;
21         uint64_t l = l1 * l2, m = l1 * h2 + l2 * h1, h = h1 * h2;
22         uint64_t ret = (l & md) + (l >> 61) + (h << 3) + (m >> 29) + (m << 35
23             >> 3) + 1;
24         ret = (ret & md) + (ret >> 61);
25         ret = (ret & md) + (ret >> 61);
26         return ret - 1;
27     }
28
29     void ensure_pw(int sz) {
30         int cur = (int) pw.size();
31         if (cur < sz) {
32             pw.resize(sz);
33             for (int i = cur; i < sz; i++) {
34                 pw[i] = mulmod(pw[i - 1], step);
35             }
36         }
37     }
38 }

```

```

34     }
35 }
36
37 vector<uint64_t> pref;
38 int n;
39
40 template<typename T>
41 hash61(const T& s) {
42     n = (int) s.size();
43     ensure_pw(n + 1);
44     pref.resize(n + 1);
45     pref[0] = 1;
46     for (int i = 0; i < n; i++) {
47         pref[i + 1] = addmod(mulmod(pref[i], step), s[i]);
48     }
49 }
50
51 inline uint64_t operator()(const int from, const int to) const {
52     assert(0 <= from && from <= to && to <= n - 1);
53     return submod(pref[to + 1], mulmod(pref[from], pw[to - from + 1]));
54 }
55 };
56
57 uint64_t hash61::step = (md >> 2) + rng() % (md >> 1);
58 vector<uint64_t> hash61::pw = vector<uint64_t>(1, 1);

```

6.4 kmp.cpp

```

1 template <typename T>
2 vector<int> kmp_table(int n, const T &s) {
3     vector<int> p(n, 0);
4     int k = 0;
5     for (int i = 1; i < n; i++) {
6         while (k > 0 && !(s[i] == s[k])) {
7             k = p[k - 1];
8         }
9         if (s[i] == s[k]) {
10             k++;
11         }
12         p[i] = k;
13     }

```

```

14     return p;
15 }
16
17 template <typename T>
18 vector<int> kmp_table(const T &s) {
19     return kmp_table((int) s.size(), s);
20 }
21
22 template <typename T>
23 vector<int> kmp_search(int n, const T &s, int m, const T &w, const vector<
    int> &p) {
24     assert(n >= 1 && (int) p.size() == n);
25     vector<int> res;
26     int k = 0;
27     for (int i = 0; i < m; i++) {
28         while (k > 0 && (k == n || !(w[i] == s[k]))) {
29             k = p[k - 1];
30         }
31         if (w[i] == s[k]) {
32             k++;
33         }
34         if (k == n) {
35             res.push_back(i - n + 1);
36         }
37     }
38     return res;
39     // returns 0-indexed positions of occurrences of s in w
40 }
41
42 template <typename T>
43 vector<int> kmp_search(const T &s, const T &w, const vector<int> &p) {
44     return kmp_search((int) s.size(), s, (int) w.size(), w, p);
45 }

```

6.5 manacher.cpp

```

1 template <typename T>
2 vector<int> manacher(int n, const T &s) {
3     if (n == 0) {
4         return vector<int>();
5     }

```



```

6   vector<int> res(2 * n - 1, 0);
7   int l = -1, r = -1;
8   for (int z = 0; z < 2 * n - 1; z++) {
9       int i = (z + 1) >> 1;
10      int j = z >> 1;
11      int p = (i >= r ? 0 : min(r - i, res[2 * (l + r) - z]));
12      while (j + p + 1 < n && i - p - 1 >= 0) {
13          if (!(s[j + p + 1] == s[i - p - 1])) {
14              break;
15          }
16          p++;
17      }
18      if (j + p > r) {
19          l = i - p;
20          r = j + p;
21      }
22      res[z] = p;
23  }
24  return res;
25  // res[2 * i] = odd radius in position i
26  // res[2 * i + 1] = even radius between positions i and i + 1
27  // s = "abaa" -> res = {0, 0, 1, 0, 0, 1, 0}
28  // in other words, for every z from 0 to 2 * n - 2:
29  // calculate i = (z + 1) >> 1 and j = z >> 1
30  // now there is a palindrome from i - res[z] to j + res[z]
31  // (watch out for i > j and res[z] = 0)
32 }
33
34 template <typename T>
35 vector<int> manacher(const T &s) {
36     return manacher((int) s.size(), s);
37 }

```

6.6 suffix-array.cpp

```

1  template <typename T>
2  vector<int> suffix_array(int n, const T &s, int char_bound) {
3      vector<int> a(n);
4      if (n == 0) {
5          return a;
6      }

```

```

7      if (char_bound != -1) {
8          vector<int> aux(char_bound, 0);
9          for (int i = 0; i < n; i++) {
10              aux[s[i]]++;
11          }
12          int sum = 0;
13          for (int i = 0; i < char_bound; i++) {
14              int add = aux[i];
15              aux[i] = sum;
16              sum += add;
17          }
18          for (int i = 0; i < n; i++) {
19              a[aux[s[i]]++] = i;
20          }
21      } else {
22          iota(a.begin(), a.end(), 0);
23          sort(a.begin(), a.end(), [&s](int i, int j) { return s[i] < s[j]; });
24      }
25      vector<int> sorted_by_second(n);
26      vector<int> ptr_group(n);
27      vector<int> new_group(n);
28      vector<int> group(n);
29      group[a[0]] = 0;
30      for (int i = 1; i < n; i++) {
31          group[a[i]] = group[a[i - 1]] + (s[a[i]] == s[a[i - 1]]);
32      }
33      int cnt = group[a[n - 1]] + 1;
34      int step = 1;
35      while (cnt < n) {
36          int at = 0;
37          for (int i = n - step; i < n; i++) {
38              sorted_by_second[at++] = i;
39          }
40          for (int i = 0; i < n; i++) {
41              if (a[i] - step >= 0) {
42                  sorted_by_second[at++] = a[i] - step;
43              }
44          }
45          for (int i = n - 1; i >= 0; i--) {
46              ptr_group[group[a[i]]] = i;
47          }
48          for (int i = 0; i < n; i++) {

```

```

49     int x = sorted_by_second[i];
50     a[ptr_group[group[x]]++] = x;
51 }
52 new_group[a[0]] = 0;
53 for (int i = 1; i < n; i++) {
54     if (group[a[i]] != group[a[i - 1]]) {
55         new_group[a[i]] = new_group[a[i - 1]] + 1;
56     } else {
57         int pre = (a[i - 1] + step >= n ? -1 : group[a[i - 1] + step]);
58         int cur = (a[i] + step >= n ? -1 : group[a[i] + step]);
59         new_group[a[i]] = new_group[a[i - 1]] + (pre != cur);
60     }
61 }
62 swap(group, new_group);
63 cnt = group[a[n - 1]] + 1;
64 step <= 1;
65 }
66 return a;
67 }
68
69 template <typename T>
70 vector<int> suffix_array(const T &s, int char_bound) {
71     return suffix_array((int) s.size(), s, char_bound);
72 }
73
74 template <typename T>
75 vector<int> build_lcp(int n, const T &s, const vector<int> &sa) {
76     assert((int) sa.size() == n);
77     vector<int> pos(n);
78     for (int i = 0; i < n; i++) {
79         pos[sa[i]] = i;
80     }
81     vector<int> lcp(max(n - 1, 0));
82     int k = 0;
83     for (int i = 0; i < n; i++) {
84         k = max(k - 1, 0);
85         if (pos[i] == n - 1) {
86             k = 0;
87         } else {
88             int j = sa[pos[i] + 1];
89             while (i + k < n && j + k < n && s[i + k] == s[j + k]) {
90                 k++;

```

```

91     }
92     lcp[pos[i]] = k;
93 }
94 }
95 return lcp;
96 }
97
98 template <typename T>
99 vector<int> build_lcp(const T &s, const vector<int> &sa) {
100     return build_lcp((int) s.size(), s, sa);
101 }

```

6.7 z.cpp

```

1  template <typename T>
2  vector<int> z_function(int n, const T &s) {
3      vector<int> z(n, 0);
4      int l = 0, r = 0;
5      for (int i = 1; i < n; i++) {
6          z[i] = (i > r ? 0 : min(r - i + 1, z[i - l]));
7          while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
8              z[i]++;
9          }
10         if (i + z[i] - 1 > r) {
11             l = i;
12             r = i + z[i] - 1;
13         }
14     }
15     return z;
16 }
17
18 template <typename T>
19 vector<int> z_function(const T &s) {
20     return z_function((int) s.size(), s);
21 }

```

7 template

7.1 hc.cpp

```

1  /**
2   *   author:   tourist
3   *   created:  $CURRENT_DATE.$CURRENT_MONTH.$CURRENT_YEAR $CURRENT_HOUR:
               $CURRENT_MINUTE:$CURRENT_SECOND
4  **/
5  #include <bits/stdc++.h>
6
7  using namespace std;
8
9  #ifdef LOCAL
10 #include "algo/debug.h"
11 #else
12 #define debug(...) 42
13 #endif
14
15 int main() {
16     ios::sync_with_stdio(false);
17     cin.tie(0);
18     int tt;
19     cin >> tt;
20     for (int qq = 1; qq <= tt; qq++) {
21         cout << "Case_" << qq << ":_";
22         ${0}
23     }
24     return 0;
25 }
```

7.2 multithreaded.cpp

```

1  /**
2   *   author:   tourist
3   *   created:  $CURRENT_DATE.$CURRENT_MONTH.$CURRENT_YEAR $CURRENT_HOUR:
               $CURRENT_MINUTE:$CURRENT_SECOND
4  **/
5  #include <bits/stdc++.h>
6
7  using namespace std;
8
9  class Solution {
10 public:
11     int k;
```

```

12     string s, w;
13
14     void readData() {
15
16     }
17
18     void solve(stringstream& out) {
19
20     }
21 };
22
23 const int maxThreads = 8;
24 const int numTests = 1000;
25
26 stringstream out[numTests];
27 mutex mu;
28 int cur, tt;
29 thread threads[maxThreads];
30
31 void solutionRunner() {
32     while (true) {
33         Solution s;
34         int id;
35         mu.lock();
36         if (cur >= tt) {
37             mu.unlock();
38             return;
39         }
40         id = cur;
41         cur++;
42         s.readData();
43         mu.unlock();
44         s.solve(out[id]);
45     }
46 }
47
48 using namespace std::chrono;
49
50 long long now() {
51     milliseconds ms = duration_cast<milliseconds>(system_clock::now().
               time_since_epoch());
52     return ms.count();
```

```

53 }
54
55 int main() {
56     ios::sync_with_stdio(false);
57     cin.tie(0);
58     long long start = now();
59     cin >> tt;
60     cur = 0;
61     for (int i = 0; i < maxThreads; i++) {
62         threads[i] = thread(solutionRunner);
63     }
64     for (int i = 0; i < maxThreads; i++) {
65         threads[i].join();
66     }
67     for (int i = 0; i < tt; i++) {
68         cout << "Case_" << i + 1 << ":_" << '\n';
69         cout << out[i].str();
70     }
71     cerr << "time_=" << now() - start << "_ms" << endl;
72     return 0;
73 }

```

7.3 q1.cpp

```

1 /**
2  *   author:   tourist
3  *   created:  $CURRENT_DATE.$CURRENT_MONTH.$CURRENT_YEAR $CURRENT_HOUR:
               $CURRENT_MINUTE:$CURRENT_SECOND
4  */
5 #include <bits/stdc++.h>
6
7 using namespace std;
8
9 #ifdef LOCAL
10 #include "algo/debug.h"
11 #else
12 #define debug(...) 42

```

```

13 #endif
14
15 int main() {
16     ios::sync_with_stdio(false);
17     cin.tie(0);
18     ${0}
19     return 0;
20 }

```

7.4 qt.cpp

```

1 /**
2  *   author:   tourist
3  *   created:  $CURRENT_DATE.$CURRENT_MONTH.$CURRENT_YEAR $CURRENT_HOUR:
               $CURRENT_MINUTE:$CURRENT_SECOND
4  */
5 #include <bits/stdc++.h>
6
7 using namespace std;
8
9 #ifdef LOCAL
10 #include "algo/debug.h"
11 #else
12 #define debug(...) 42
13 #endif
14
15 int main() {
16     ios::sync_with_stdio(false);
17     cin.tie(0);
18     int tt;
19     cin >> tt;
20     while (tt--) {
21         ${0}
22     }
23     return 0;
24 }

```