



河南大學  
Henan University

# Dissonant Interval

Contestant

曹家宁  
Jianing Cao

罗施达  
Shida Luo

孙小文  
Xiaowen Sun

目录

<b>1</b>	<b>图论</b>	<b>1</b>			
1.1	EulerianPath	1			
1.2	Ex-Kruskal	1			
1.3	MinCircuit	3			
1.4	TopoSort	4			
1.5	TreeCentroid	4			
1.6	TreeDiameter	4			
1.7	全源最短路	4			
1.7.1	Floyd	4			
1.7.2	Johnson	5			
1.8	单源最短路	6			
1.8.1	Dijkstra	6			
1.8.2	SPFA	6			
1.9	图的匹配	6			
1.9.1	AugmentingPath	6			
1.9.2	Kuhn-Munkres	7			
1.10	树哈希	9			
1.10.1	TreeHashing-Rooted	9			
1.10.2	TreeHashing-Unrooted	9			
1.11	树链剖分	10			
1.11.1	HLD-Edge	10			
1.11.2	HLD-Vertex	12			
1.11.3	HLD	14			
1.12	环计数	15			
1.12.1	CommonRing	15			
1.12.2	QuarternaryRing	15			
1.12.3	TernaryRing	16			
1.13	生成树	16			
1.13.1	DMST	16			
1.13.2	Kruskal	18			
1.13.3	SSST	19			
1.13.4	SteinerTree	22			
1.14	矩阵树定理	23			
1.14.1	MatrixTree-Directed	23			
1.14.2	MatrixTree-Undirected	25			
1.15	缩点	27			
1.15.1	EBCC	27			
1.15.2	SCC	27			
1.16	网络流	28			
1.16.1	Dinic	28			
1.16.2	ISAP	29			
1.16.3	Primal-Dual	30			
1.16.4	Stoer-Wagner	31			
<b>2</b>	<b>字符串</b>	<b>32</b>			
2.1	Duo-Hashing	32			
2.2	KMP	32			
2.3	Manacher	33			
2.4	Z-Function	33			
<b>3</b>	<b>数学</b>	<b>33</b>			
3.1	CatalanNumbers	33			
3.2	Combination-Dynamic	33			
3.3	Combination	34			
3.4	Matrix	35			
3.5	Power-Inv	36			
3.6	Xor	36			
<b>4</b>	<b>数据结构</b>	<b>37</b>			
4.1	01-Trie	37			

4.2	LeftistTree . . . . .	37
4.3	SparseTableByEnar . . . . .	39
4.4	并查集 . . . . .	39
4.4.1	DSU-Rollback . . . . .	39
4.4.2	DSU . . . . .	39
4.5	树状数组 . . . . .	40
4.5.1	Fenwick-Range . . . . .	40
4.5.2	Fenwick . . . . .	40
4.6	线段树 . . . . .	40
4.6.1	LazySegmentTree-Extra . . . . .	40
4.6.2	LazySegmentTree-Short . . . . .	42
4.6.3	LazySegmentTree . . . . .	42
4.6.4	PersistentSegmentTreeByEnar . . . . .	43
<b>5</b>	<b>数论</b>	<b>45</b>
5.1	CRT . . . . .	45
5.2	Euclidean-Like . . . . .	45
5.3	Ex-GCD . . . . .	46
5.4	Factorize . . . . .	46
5.5	GCD-LCM . . . . .	47
5.6	LCE . . . . .	47
5.7	Miller-Rabin . . . . .	47
5.8	Phi . . . . .	47
5.9	Pollard-Rho . . . . .	48
5.10	Sieve . . . . .	48
<b>6</b>	<b>杂类</b>	<b>48</b>
6.1	QuickRead . . . . .	48
<b>7</b>	<b>计算几何</b>	<b>48</b>
7.1	BinaryHull . . . . .	48
7.2	ConvexHull . . . . .	49

7.3	Line . . . . .	50
7.4	RotatingCalipers . . . . .	51
7.5	Segment . . . . .	52
7.6	Vecteur . . . . .	53

# 1 图论

## 1.1 EulerianPath

```
1 struct Edge {
2     bool flag;
3     i32 v, pos;
4     bool operator < (const Edge &u) const {
5         return v < u.v;
6     }
7 };
8
9 auto hierholzer(const std::vector<std::vector<i32>> &adj) {
10     i32 n = adj.size();
11     std::vector<std::vector<Edge>> nadj(n);
12     for (i32 u = 0; u < n; ++u) {
13         for (const auto &v : adj[u]) {
14             nadj[u].push_back({true, v, 0});
15         }
16         std::sort(nadj[u].begin(), nadj[u].end());
17     }
18     std::vector<i32> top(n), deg(n);
19     for (i32 u = 0; u < n; ++u) {
20         for (auto &[flag, v, pos] : nadj[u]) {
21             pos = top[v]++;
22         }
23         deg[u] = adj[u].size();
24     }
25     std::vector<i32> cnt(n), res;
26     auto dfs = [&](auto self, i32 u) -> void {
27         i32 cur = cnt[u];
28         while (cur < adj[u].size()) {
29             auto [flag, v, pos] = nadj[u][cur];
30             if (flag) {
31                 nadj[u][cur].flag = false;
32                 nadj[v][pos].flag = false;
33                 cur++;
34                 self(self, v);
35             } else {
36                 cur++;
37             }
38         }
39         res.push_back(u);
40     };
41     i32 st = 0;
42     for (i32 i = 0; i < n; ++i) {
43         if (!deg[st] && deg[i]) {
44             st = i;
```

```
45         } else if (!(deg[st] & 1) && (deg[i] & 1)) {
46             st = i;
47         }
48     }
49     dfs(dfs, st);
50     std::reverse(res.begin(), res.end());
51     return res;
52 }
```

## 1.2 Ex-Kruskal

```
1 struct DSU {
2     std::vector<i32> p, siz;
3     DSU() {}
4     DSU(i32 n) {
5         init(n);
6     }
7     void init(i32 n) {
8         p.resize(n);
9         siz.assign(n, 1);
10        std::iota(p.begin(), p.end(), 0);
11    }
12    i32 find(i32 x) {
13        while (x != p[x]) x = p[x] = p[p[x]];
14        return x;
15    }
16    bool same(i32 x, i32 y) {
17        return find(x) == find(y);
18    }
19    bool merge(i32 x, i32 y) {
20        x = find(x);
21        y = find(y);
22        if (x == y) {
23            return false;
24        } else {
25            p[y] = x;
26            siz[x] += siz[y];
27            return true;
28        }
29    }
30    i32 size(i32 x) {
31        return siz[find(x)];
32    }
33 };
34
35 struct HLD {
36     i32 n, cur;
37     std::vector<i32> siz, top, dep, parent, in, out, seq;
```

```

38 std::vector<std::vector<i32>> adj;
39 HLD() {}
40 HLD(i32 n) {
41     init(n);
42 }
43 void init(i32 n) {
44     this->n = n;
45     in.resize(n);
46     out.resize(n);
47     siz.resize(n);
48     top.resize(n);
49     dep.resize(n);
50     seq.resize(n);
51     parent.resize(n);
52     adj.assign(n, {});
53     cur = 0;
54 }
55 void addEdge(i32 u, i32 v) {
56     adj[u].push_back(v);
57     adj[v].push_back(u);
58 }
59 void work(i32 root = 0) {
60     top[root] = root;
61     dep[root] = 0;
62     parent[root] = -1;
63     dfs1(root);
64     dfs2(root);
65 }
66 void dfs1(i32 u) {
67     if (parent[u] != -1) {
68         adj[u].erase(std::find(adj[u].begin(), adj[u].end(), parent[u]));
69     }
70     siz[u] = 1;
71     for (auto &v : adj[u]) {
72         parent[v] = u;
73         dep[v] = dep[u] + 1;
74         dfs1(v);
75         siz[u] += siz[v];
76         if (siz[v] > siz[adj[u][0]]) {
77             std::swap(v, adj[u][0]);
78         }
79     }
80 }
81 void dfs2(i32 u) {
82     in[u] = cur++;
83     seq[in[u]] = u;
84     for (auto v : adj[u]) {
85         top[v] = v == adj[u][0] ? top[u] : v;

```

```

86         dfs2(v);
87     }
88     out[u] = cur;
89 }
90 i32 lca(i32 u, i32 v) {
91     while (top[u] != top[v]) {
92         if (dep[top[u]] > dep[top[v]]) {
93             u = parent[top[u]];
94         } else {
95             v = parent[top[v]];
96         }
97     }
98     return dep[u] < dep[v] ? u : v;
99 }
100 i32 rootedLca(i32 a, i32 b, i32 c) {
101     return lca(a, b) ^ lca(b, c) ^ lca(c, a);
102 }
103 };
104
105 struct Node {
106     i32 u, v;
107     i64 w;
108     bool operator < (const Node &u) const {
109         return w < u.w;
110     }
111     bool operator > (const Node &u) const {
112         return w > u.w;
113     }
114 };
115
116 template <typename Func = std::function<bool(const Node&, const Node&>>
117 struct Kruskal {
118     Func cmp;
119     i32 n;
120     HLD tree;
121     DSU copo;
122     std::vector<i64> val;
123     Kruskal(i32 n, std::vector<Node> &edg, Func cmp) : n(n), cmp(cmp) {
124         val.resize(2 * n - 1);
125         tree.init(2 * n - 1);
126         copo.init(2 * n - 1);
127         build(edg);
128     }
129     void build(std::vector<Node> &edg) {
130         std::sort(edg.begin(), edg.end(), cmp);
131         i32 m = edg.size();
132         i32 cur = n;
133         for (i32 i = 0; i < m; ++i) {

```

```

134     auto [u, v, w] = edg[i];
135     u = copo.find(u);
136     v = copo.find(v);
137     if (u != v) {
138         copo.merge(cur, u);
139         copo.merge(cur, v);
140         tree.addEdge(cur, u);
141         tree.addEdge(cur, v);
142         val[cur] = w;
143         cur++;
144         if (cur == 2 * n - 1) {
145             break;
146         }
147     }
148 }
149 std::vector<bool> vis(2 * n - 1);
150 for (i32 i = 0; i < n; ++i) {
151     i32 root = copo.find(i);
152     if (!vis[root]) {
153         tree.work(root);
154         vis[root] = true;
155     }
156 }
157 }
158 bool same(i32 u, i32 v) {
159     return copo.same(u, v);
160 }
161 i32 lca(i32 u, i32 v) {
162     if (copo.same(u, v)) {
163         return tree.lca(u, v);
164     } else {
165         return -1;
166     }
167 }
168 i64 bnPath(i32 u, i32 v) {
169     if (copo.same(u, v)) {
170         return val[tree.lca(u, v)];
171     } else {
172         return -1;
173     }
174 }
175 };
176
177 using greater = std::greater<Node>;
178 using less = std::less<Node>;

```

### 1.3 MinCircuit

```

1  const i64 INF = 1E18;
2
3  struct Edge {
4      i32 u, v;
5      i64 w;
6  };
7
8  std::vector<std::vector<i64>> dist;
9  std::vector<i32> path;
10
11 i64 floyd(i32 n, const std::vector<Edge> &edg) {
12     std::vector val(n, std::vector<i64>(n, INF));
13     dist = std::vector(n, std::vector<i64>(n, INF));
14     for (const auto &[u, v, w] : edg) {
15         dist[u][v] = std::min(dist[u][v], w);
16         dist[v][u] = std::min(dist[v][u], w);
17         val[u][v] = val[v][u] = w;
18     }
19     for (i32 i = 0; i < n; ++i) {
20         dist[i][i] = 0;
21     }
22     std::vector pos(n, std::vector<i64>(n, -1));
23     auto dfs = [&](auto self, i32 u, i32 v) -> void {
24         if (pos[u][v] == -1) return;
25         i32 k = pos[u][v];
26         self(self, u, k);
27         path.push_back(k);
28         self(self, k, v);
29     };
30     res = INF;
31     for (i32 k = 0; k < n; ++k) {
32         for (i32 u = 0; u < k; ++u) {
33             for (i32 v = 0; v < u; ++v) {
34                 i64 cur = dist[u][v] + val[u][k] + val[k][v];
35                 if (res > cur) {
36                     res = cur;
37                     path.clear();
38                     path.push_back(u);
39                     path.push_back(k);
40                     path.push_back(v);
41                     dfs(dfs, v, u);
42                 }
43             }
44         }
45         for (i32 u = 0; u < n; ++u) {
46             for (i32 v = 0; v < n; ++v) {
47                 i64 cur = dist[u][k] + dist[k][v];

```

```

48         if (dist[u][v] > cur) {
49             dist[u][v] = cur;
50             pos[u][v] = k;
51         }
52     }
53 }
54 }
55 return res >= INF ? -1 : res;
56 }

```

## 1.4 TopoSort

```

1 auto topo(std::vector<std::vector<i32>> &adj) {
2     i32 n = adj.size();
3     std::vector<i32> in(n);
4     for (i32 i = 0; i < n; ++i) {
5         for (const auto &u : adj[i]) {
6             in[u]++;
7         }
8     }
9     std::vector<i32> res;
10    std::queue<i32> q;
11    for (i32 i = 0; i < n; ++i) {
12        if (in[i] == 0) {
13            q.push(i);
14        }
15    }
16    while (!q.empty()) {
17        auto u = q.front();
18        q.pop();
19        res.push_back(u);
20        for (const auto &v : adj[u]) {
21            in[v]--;
22            if (in[v] == 0) {
23                q.push(v);
24            }
25        }
26    }
27    return res;
28 }

```

## 1.5 TreeCentroid

```

1 std::vector<i32> siz, wei;
2
3 auto getCentroid(const std::vector<std::vector<i32>> &adj) {
4     i32 n = adj.size();

```

```

5     siz.assign(n, 1);
6     wei.assign(n, 0);
7     std::array<i32, 2> cen{-1, -1};
8     auto dfs = [&](auto self, i32 u, i32 p) -> void {
9         for (const auto &v : adj[u]) {
10             if (v == p) continue;
11             self(self, v, u);
12             wei[u] = std::max(wei[u], wei[v]);
13             siz[u] += siz[v];
14         }
15         wei[u] = std::max(wei[u], n - siz[u]);
16         if (wei[u] <= n / 2) {
17             cen[cen[0] != -1] = u;
18         }
19     };
20     return cen;
21 }

```

## 1.6 TreeDiameter

```

1 i32 getDiameter(const std::vector<std::vector<i32>> &adj) {
2     i32 d = 0;
3     i32 n = adj.size();
4     std::vector<i32> dp(n);
5     auto dfs = [&](auto self, i32 u, i32 p) -> void {
6         for (const auto &v : adj[u]) {
7             if (v == p) continue;
8             self(self, v, u);
9             d = std::max(d, dp[u] + dp[v] + 1);
10            dp[u] = std::max(dp[u], dp[v] + 1);
11        }
12    };
13    dfs(dfs, 0, -1);
14    return d;
15 }

```

## 1.7 全源最短路

### 1.7.1 Floyd

```

1 const i64 INF = 1E18;
2
3 struct Edge {
4     i32 u, v;
5     i64 w;
6 };
7

```

```

8 std::vector<std::vector<i64>> dist;
9
10 void floyd(i32 n, const std::vector<Edge> &edg) {
11     dist = std::vector(n, std::vector<i64>(n, INF));
12     for (const auto &[u, v, w] : edg) {
13         dist[u][v] = std::min(dist[u][v], w);
14         dist[v][u] = std::min(dist[v][u], w);
15     }
16     for (i32 i = 0; i < n; ++i) {
17         dist[i][i] = 0;
18     }
19     for (i32 k = 0; k < n; ++k) {
20         for (i32 u = 0; u < n; ++u) {
21             for (i32 v = 0; v < n; ++v) {
22                 dist[u][v] = std::min(dist[u][v], dist[u][k] + dist[k][v]);
23             }
24         }
25     }
26 }

```

### 1.7.2 Johnson

```

1 const i64 INF = 1E18;
2
3 struct Node {
4     i32 v;
5     i64 w;
6     bool operator < (const Node &u) const {
7         return w > u.w;
8     }
9 };
10
11 std::vector<i64> pot;
12 std::vector<std::vector<i64>> dist;
13
14 bool spfa(i32 s, std::vector<std::vector<Node>> &adj) {
15     i32 n = adj.size();
16     std::vector<bool> vis(n);
17     std::vector<i32> cnt(n);
18     pot.assign(n, INF);
19     vis[s] = true;
20     pot[s] = 0;
21     std::queue<i32> q;
22     q.push(s);
23     while (!q.empty()) {
24         i32 u = q.front();
25         q.pop();
26         vis[u] = false;

```

```

27         for (const auto &[v, w] : adj[u]) {
28             if (pot[v] > pot[u] + w) {
29                 pot[v] = pot[u] + w;
30                 cnt[v] = cnt[u] + 1;
31                 if (cnt[v] > n) {
32                     return false;
33                 }
34                 if (!vis[v]) {
35                     vis[v] = true;
36                     q.push(v);
37                 }
38             }
39         }
40     }
41     return true;
42 }
43
44 void dijkstra(i32 s, std::vector<std::vector<Node>> &adj) {
45     i32 n = adj.size();
46     dist[s].assign(n, INF);
47     std::vector<bool> vis(n);
48     std::priority_queue<Node> pq;
49     pq.push({s, 0});
50     dist[s][s] = 0;
51     while (!pq.empty()) {
52         auto [u, cur] = pq.top();
53         pq.pop();
54         if (vis[u]) continue;
55         vis[u] = true;
56         for (const auto &[v, w] : adj[u]) {
57             i64 nxt = cur + w;
58             if (dist[s][v] <= nxt) continue;
59             pq.push({v, nxt});
60             dist[s][v] = nxt;
61         }
62     }
63 }
64
65 bool johnson(std::vector<std::vector<Node>> &adj) {
66     i32 n = adj.size();
67     adj.push_back({});
68     for (i32 i = 0; i < n; ++i) {
69         adj[n].push_back({i, 0});
70     }
71     bool flag = spfa(n, adj);
72     adj.pop_back();
73     if (!flag) {
74         return false;

```



```

75     }
76     for (i32 u = 0; u < n; ++u) {
77         for (auto &[v, w] : adj[u]) {
78             w = w + pot[u] - pot[v];
79         }
80     }
81     dist.assign(n, std::vector<i64>(n));
82     for (i32 i = 0; i < n; ++i) {
83         dijkstra(i, adj);
84         for (i32 j = 0; j < n; ++j) {
85             if (dist[i][j] == INF) continue;
86             dist[i][j] += pot[j] - pot[i];
87         }
88     }
89     return true;
90 }

```

## 1.8 单源最短路

### 1.8.1 Dijkstra

```

1  const i64 INF = 1E18;
2
3  struct Node {
4      i32 v;
5      i64 w;
6      bool operator < (const Node &u) const {
7          return w > u.w;
8      }
9  };
10
11 std::vector<i64> dist;
12
13 void dijkstra(i32 s, std::vector<std::vector<Node>> &adj) {
14     i32 n = adj.size();
15     std::vector<bool> vis(n);
16     dist.assign(n, INF);
17     std::priority_queue<Node> pq;
18     pq.push({s, 0});
19     dist[s] = 0;
20     while (!pq.empty()) {
21         auto [u, cur] = pq.top();
22         pq.pop();
23         if (vis[u]) continue;
24         vis[u] = true;
25         for (const auto &[v, w] : adj[u]) {
26             i64 nxt = cur + w;
27             if (dist[v] <= nxt) continue;

```

```

28             pq.push({v, nxt});
29             dist[v] = nxt;
30         }
31     }
32 }

```

### 1.8.2 SPFA

```

1  const i64 INF = 1E18;
2
3  struct Node {
4      i32 v, w;
5  };
6
7  std::vector<bool> vis;
8  std::vector<i64> dist;
9  std::vector<i32> cnt;
10
11 bool spfa(i32 s, std::vector<std::vector<Node>> &adj) {
12     i32 n = adj.size();
13     vis.assign(n, false);
14     cnt.assign(n, 0);
15     dist.assign(n, INF);
16     vis[s] = true;
17     dist[s] = 0;
18     std::queue<i32> q;
19     q.push(s);
20     while (!q.empty()) {
21         i32 u = q.front();
22         q.pop();
23         vis[u] = false;
24         for (const auto &[v, w] : adj[u]) {
25             if (dist[v] > dist[u] + w) {
26                 dist[v] = dist[u] + w;
27                 cnt[v] = cnt[u] + 1;
28                 if (cnt[v] > n) {
29                     return false;
30                 }
31                 if (!vis[v]) {
32                     vis[v] = true;
33                     q.push(v);
34                 }
35             }
36         }
37     }
38     return true;
39 }

```

## 1.9 图的匹配

### 1.9.1 AugmentingPath

```
1 const i64 INF = 1E18;
2
3 struct Edge {
4     i32 u, v;
5     i64 cap, flow;
6 };
7
8 struct Dinic {
9     i32 n, s, t;
10    std::vector<Edge> edg;
11    std::vector<i32> dep, cur;
12    std::vector<std::vector<i32>> pos;
13    Dinic(i32 n) : n(n) {
14        dep.resize(n);
15        cur.resize(n);
16        pos.resize(n);
17    }
18    void addEdge(i32 u, i32 v, i64 w) {
19        edg.push_back({u, v, w, 0});
20        edg.push_back({v, u, 0, 0});
21        i32 m = edg.size();
22        pos[u].push_back(m - 2);
23        pos[v].push_back(m - 1);
24    }
25    i64 bfs() {
26        std::vector<bool> vis(n);
27        std::queue<i32> q;
28        q.push(s);
29        dep[s] = 0;
30        vis[s] = true;
31        while (!q.empty()) {
32            i32 now = q.front();
33            q.pop();
34            for (int i = 0; i < pos[now].size(); i++) {
35                auto &[u, v, cap, flow] = edg[pos[now][i]];
36                if (!vis[v] && cap > flow) {
37                    dep[v] = dep[u] + 1;
38                    vis[v] = true;
39                    q.push(v);
40                }
41            }
42        }
43        return vis[t];
44    }
45    i64 dfs(i32 now, i64 ctn) {
```

```
46    if (now == t || ctn == 0) {
47        return ctn;
48    }
49    i64 res = 0;
50    for (i32 i = cur[now]; i < pos[now].size(); ++i) {
51        auto &[u, v, cap, flow] = edg[pos[now][i]];
52        auto &[ru, rv, rcap, rflow] = edg[pos[now][i] ^ 1];
53        cur[now] = i;
54        if (dep[v] == dep[u] + 1 && cap > flow) {
55            i64 aug = dfs(v, std::min(ctn - res, cap - flow));
56            if (aug > 0) {
57                res += aug;
58                flow += aug;
59                rflow -= aug;
60                if (res == ctn) {
61                    return res;
62                }
63            }
64        }
65    }
66    return res;
67 }
68 i64 maxFlow(i32 s, i32 t) {
69     this->s = s;
70     this->t = t;
71     i64 res = 0;
72     while (bfs()) {
73         cur.assign(n, 0);
74         res += dfs(s, INF);
75     }
76     return res;
77 }
78 };
79
80 struct AugmentingPath : Dinic {
81     std::unordered_set<i32> l, r;
82     AugmentingPath(i32 n) : Dinic(n + 2) {}
83     void addEdge(i32 u, i32 v) {
84         Dinic::addEdge(u, v, 1);
85         l.insert(u);
86         r.insert(v);
87     }
88     i64 match() {
89         i32 s = n - 1;
90         i32 t = n - 2;
91         for (const auto &u : l) {
92             Dinic::addEdge(s, u, 1);
93         }
```

```

94     for (const auto &u : r) {
95         Dinic::addEdge(u, t, 1);
96     }
97     return Dinic::maxFlow(s, t);
98 }
99 };

```

### 1.9.2 Kuhn-Munkres

```

1  const i64 INF = 1E18;
2
3  struct KuhnMunkres {
4      i32 n, l, r;
5      std::queue<i32> q;
6      std::vector<i32> mchl, mchr, pre;
7      std::vector<i64> labl, labr, slk;
8      std::vector<bool> visl, visr;
9      std::vector<std::vector<i64>> wei;
10     KuhnMunkres(i32 l, i32 r) : l(l), r(r), n(std::max(l, r)) {
11         wei.assign(n, std::vector<i64>(n, 0LL));
12         mchl.assign(n, -1);
13         mchr.assign(n, -1);
14         pre.assign(n, -1);
15         labl.assign(n, 0);
16         labr.assign(n, 0);
17         slk.resize(n);
18         visl.resize(n);
19         visr.resize(n);
20     }
21     void addEdge(i32 u, i32 v, i64 w) {
22         wei[u][v] = std::max<i64>(0LL, w);
23     }
24     bool check(i32 idx) {
25         visl[idx] = true;
26         if (mchl[idx] != -1) {
27             q.push(mchl[idx]);
28             visr[mchl[idx]] = true;
29             return false;
30         }
31         while (idx != -1) {
32             mchl[idx] = pre[idx];
33             std::swap(idx, mchr[pre[idx]]);
34         }
35         return true;
36     }
37     bool bfs(i32 idx) {
38         while (!q.empty()) {
39             i32 v = q.front();

```

```

40         q.pop();
41         for (i32 u = 0; u < n; ++u) {
42             if (visl[u]) continue;
43             i64 d = labl[u] + labr[v] - wei[u][v];
44             if (slk[u] < d) {
45                 continue;
46             }
47             pre[u] = v;
48             if (d > 0) {
49                 slk[u] = d;
50             } else if (check(u)) {
51                 return true;
52             }
53         }
54     }
55     return false;
56 }
57 i64 match() {
58     for (i32 i = 0; i < n; ++i) {
59         labl[i] = *std::max_element(wei[i].begin(), wei[i].end());
60     }
61     for (i32 i = 0; i < n; ++i) {
62         visl.assign(n, false);
63         visr.assign(n, false);
64         slk.assign(n, INF);
65         while (!q.empty()) {
66             q.pop();
67         }
68         q.push(i);
69         visr[i] = true;
70         while (true) {
71             if (bfs(i)) {
72                 break;
73             }
74             i64 d = INF;
75             for (i32 j = 0; j < n; ++j) {
76                 if (!visl[j]) {
77                     d = std::min(d, slk[j]);
78                 }
79             }
80             for (i32 j = 0; j < n; ++j) {
81                 if (visr[j]) {
82                     labr[j] -= d;
83                 }
84                 if (visl[j]) {
85                     labl[j] += d;
86                 } else {
87                     slk[j] -= d;

```

```

88     }
89     }
90     while (!q.empty()) {
91         q.pop();
92     }
93     bool flag = false;
94     for (i32 j = 0; j < n; ++j) {
95         if (!visl[j] && !slk[j] && check(j)) {
96             flag = true;
97             break;
98         }
99     }
100     if (flag) {
101         break;
102     }
103 }
104 }
105 i64 res = 0;
106 for (i32 i = 0; i < n; ++i) {
107     if (mchl[i] == -1) {
108         continue;
109     } else if (wei[i][mchl[i]] > 0) {
110         res += wei[i][mchl[i]];
111     } else {
112         mchl[i] = -1;
113     }
114 }
115 return res;
116 }
117 };

```

## 1.10 树哈希

### 1.10.1 TreeHashing-Rooted

```

1 std::mt19937 rng(std::chrono::steady_clock::now().time_since_epoch().count());
2 const u64 MASK = rng();
3
4 struct RTHashing {
5     std::set<u64> trees;
6     std::vector<u64> hash;
7     RTHashing(std::vector<std::vector<i32>> &adj) {
8         hash.assign(adj.size(), 0);
9         std::function<void(i32, i32)> work = [&](i32 u, i32 p) {
10             for (const auto &v : adj[u]) {
11                 if (v == p) continue;
12                 work(v, u);
13                 hash[u] += shift(hash[v]);

```

```

14     }
15     trees.insert(hash[u]);
16 };
17 work(0, -1);
18 }
19 u64 shift(u64 x) {
20     x ^= MASK;
21     x ^= x << 13;
22     x ^= x >> 7;
23     x ^= x << 17;
24     x ^= MASK;
25     return x;
26 }
27 u64 getHash(i32 idx) {
28     return hash[idx];
29 }
30 i32 count() {
31     return trees.size();
32 }
33 };

```

### 1.10.2 TreeHashing-Unrooted

```

1 std::mt19937 rng(std::chrono::steady_clock::now().time_since_epoch().count());
2 const u64 MASK = rng();
3
4 struct UTHashing {
5     i32 n;
6     std::vector<u64> sub, rt;
7     std::map<u64, i64> trees;
8     std::vector<std::vector<i32>> *adj;
9     UTHashing(std::vector<std::vector<i32>> &adj) {
10         n = adj.size();
11         this->adj = &adj;
12         sub.assign(n, 0);
13         rt.assign(n, 0);
14         std::function<void(i32, i32)> getSub = [&](i32 u, i32 p) {
15             for (const auto &v : adj[u]) {
16                 if (v == p) continue;
17                 getSub(v, u);
18                 sub[u] += shift(sub[v]);
19             }
20         };
21         std::function<void(i32, i32)> getRoot = [&](i32 u, i32 p) {
22             for (const auto &v : adj[u]) {
23                 if (v == p) continue;
24                 rt[v] = sub[v] + shift(rt[u] - shift(sub[v]));
25                 getRoot(v, u);

```

```

26     }
27     trees[hash[u]] = rt[u];
28 };
29 getSub(0, -1);
30 rt[0] = sub[0];
31 getRoot(0, -1);
32 }
33 u64 shift(u64 x) {
34     x ^= MASK;
35     x ^= x << 13;
36     x ^= x >> 7;
37     x ^= x << 17;
38     x ^= MASK;
39     return x;
40 }
41 i64 count(i64 idx) {
42     return rt[idx];
43 }
44 i64 total() {
45     i64 res = 0;
46     for (const auto &[x, y] : trees) {
47         res += y;
48     }
49     return res;
50 }
51 };

```

## 1.11 树链剖分

### 1.11.1 HLD-Edge

```

1 const i64 INF = 1E18;
2 const i64 MOD = 1E9 + 7;
3
4 struct Node {
5     i32 u, v;
6     i64 w;
7     bool operator < (const Node &u) const {
8         return w < u.w;
9     }
10 };
11
12 template <typename Info, typename Tag>
13 struct HLD {
14     i32 n, cur;
15     std::vector<Tag> tag;
16     std::vector<Info> info;
17     std::vector<i32> w, id, fa, val, siz, dep, top;

```

```

18     std::vector<std::vector<i32>> adj;
19     std::vector<Node> edg;
20     HLD(i32 n) : n(n), cur(0) {
21         info.resize(4 << std::__lg(n));
22         tag.resize(4 << std::__lg(n));
23         adj.assign(n, {});
24         fa.resize(n, -1);
25         val.resize(n);
26         siz.resize(n);
27         dep.resize(n);
28         top.resize(n);
29         id.resize(n);
30         w.resize(n);
31     }
32     void add(i32 p, const Tag &k) {
33         info[p].apply(k);
34         tag[p].apply(k);
35     }
36     void update(i32 p, const Tag &k) {
37         info[p].apply(k);
38         tag[p].apply(k);
39     }
40     void pull(i32 p) {
41         info[p] = info[p << 1] + info[p << 1 | 1];
42     }
43     void push(i32 p) {
44         update(p << 1, tag[p]);
45         update(p << 1 | 1, tag[p]);
46         tag[p] = Tag();
47     }
48     void addEdge(i32 u, i32 v, i32 w) {
49         adj[u].push_back(v);
50         adj[v].push_back(u);
51         edg.push_back({u, v, w});
52     }
53     void work(i32 root = 0) {
54         top[root] = root;
55         dep[root] = 0;
56         fa[root] = -1;
57         dfs1(root);
58         dfs2(root);
59         for (auto &[u, v, w] : edg) {
60             i64 p = lca(u, v);
61             if (v == p) {
62                 val[id[u]] = w;
63             } else {
64                 val[id[v]] = w;
65             }

```

```

66     }
67     auto build = [&](auto self, i32 p, i32 l, i32 r) {
68         if(r == l) {
69             info[p].init(val[l]);
70             return;
71         }
72         i32 m = (l + r) / 2;
73         self(self, p << 1, l, m);
74         self(self, p << 1 | 1, m + 1, r);
75         pull(p);
76     };
77     build(build, 1, 0, n - 1);
78 }
79 void dfs1(i32 u) {
80     if (fa[u] != -1) {
81         adj[u].erase(std::find(adj[u].begin(), adj[u].end(), fa[u]));
82     }
83     siz[u] = 1;
84     for (auto &v : adj[u]) {
85         dep[v] = dep[u] + 1;
86         fa[v] = u;
87         dfs1(v);
88         siz[u] += siz[v];
89         if (siz[adj[u][0]] < siz[v]) {
90             std::swap(v, adj[u][0]);
91         }
92     }
93 }
94 void dfs2(i32 u) {
95     id[u] = cur;
96     val[cur++] = w[u];
97     for (const auto &v : adj[u]) {
98         if (v == fa[u]) continue;
99         top[v] = v == adj[u][0] ? top[u] : v;
100        dfs2(v);
101    }
102 }
103 i32 lca(i32 u, i32 v) {
104     while (top[u] != top[v]) {
105         if (dep[top[u]] > dep[top[v]]) {
106             u = fa[top[u]];
107         } else {
108             v = fa[top[v]];
109         }
110     }
111     return dep[u] < dep[v] ? u : v;
112 }
113 void rangeApply(i32 p, i32 x, i32 y, i32 l, i32 r, const Tag &k) {

```

```

114     if (x <= l && y >= r) {
115         add(p, k);
116         return;
117     }
118     push(p);
119     i32 m = (l + r) / 2;
120     if (x <= m) rangeApply(p << 1, x, y, l, m, k);
121     if (y > m) rangeApply(p << 1 | 1, x, y, m + 1, r, k);
122     pull(p);
123 }
124 void pathApply(i32 u, i32 v, const Tag &k) {
125     while (top[u] != top[v]) {
126         if (dep[top[u]] < dep[top[v]]) {
127             std::swap(u, v);
128         }
129         rangeApply(1, id[top[u]], id[u], 0, n - 1, k);
130         u = fa[top[u]];
131     }
132     if (dep[u] < dep[v]) {
133         std::swap(u, v);
134     }
135     if (u != v) {
136         rangeApply(1, id[v] + 1, id[u], 0, n - 1, k);
137     }
138 }
139 void treeApply(i32 u, const Tag &k) {
140     if (siz[u] > 1) {
141         rangeApply(1, id[u] + 1, id[u] + siz[u] - 1, 0, n - 1, k);
142     }
143 }
144 Info rangeQuery(i32 p, i32 x, i32 y, i32 l, i32 r) {
145     if (x <= l && y >= r) return info[p];
146     Info res{};
147     push(p);
148     i32 m = (l + r) / 2;
149     if (x <= m) res = (res + rangeQuery(p << 1, x, y, l, m));
150     if (y > m) res = (res + rangeQuery(p << 1 | 1, x, y, m + 1, r));
151     pull(p);
152     return res;
153 }
154 Info pathQuery(i32 u, i32 v) {
155     Info res{};
156     while (top[u] != top[v]) {
157         if (dep[top[u]] < dep[top[v]]) {
158             std::swap(u, v);
159         }
160         res = res + rangeQuery(1, id[top[u]], id[u], 0, n - 1);
161         u = fa[top[u]];

```

```

162     }
163     if (dep[u] < dep[v]) {
164         std::swap(u, v);
165     }
166     if (u != v) {
167         res = res + rangeQuery(1, id[v] + 1, id[u], 0, n - 1);
168     }
169     return res;
170 }
171 Info treeQuery(i32 u) {
172     if (siz[u] > 1) {
173         return rangeQuery(1, id[u] + 1, id[u] + siz[u] - 1, 0, n - 1);
174     } else {
175         return Info{};
176     }
177 }
178 };
179
180 struct Tag {
181     i64 add = 0;
182     void apply(const Tag &v) {
183         add = add + v.add;
184     }
185 };
186
187 struct Info {
188     i64 sum = 0, len = 1;
189     std::array<i64, 2> mn{INF, INF};
190     std::array<i64, 2> mx{-INF, -INF};
191     void init(const i64 &x) {
192         mn[0] = mx[0] = x;
193         sum = x;
194     }
195     void apply(const Tag &v) {
196         sum += len * v.add;
197         mn[0] += v.add;
198         mx[0] += v.add;
199         mn[1] += v.add;
200         mx[1] += v.add;
201     }
202     Info operator + (const Info &a) {
203         Info res;
204         if (mn[0] < a.mn[0]) {
205             res.mn[0] = mn[0];
206             res.mn[1] = std::min({mn[1], a.mn[0], a.mn[1]});
207         } else if (mn[0] > a.mn[0]) {
208             res.mn[0] = a.mn[0];
209             res.mn[1] = std::min({mn[0], mn[1], a.mn[1]});

```

```

210     } else {
211         res.mn[0] = mn[0];
212         res.mn[1] = std::min(mn[1], a.mn[1]);
213     }
214     if (mx[0] > a.mx[0]) {
215         res.mx[0] = mx[0];
216         res.mx[1] = std::max({mx[1], a.mx[0], a.mx[1]});
217     } else if (mx[0] < a.mx[0]) {
218         res.mx[0] = a.mx[0];
219         res.mx[1] = std::max({mx[0], mx[1], a.mx[1]});
220     } else {
221         res.mx[0] = mx[0];
222         res.mx[1] = std::max(mx[1], a.mx[1]);
223     }
224     res.sum = sum + a.sum;
225     res.len = len + a.len;
226     return res;
227 }
228 };

```

### 1.11.2 HLD-Vertex

```

1  const i64 INF = 1E18;
2  const i64 MOD = 1E9 + 7;
3
4  template <typename Info, typename Tag>
5  struct HLD {
6      i32 n, cur;
7      std::vector<Tag> tag;
8      std::vector<Info> info;
9      std::vector<i32> w, id, fa, val, siz, dep, top;
10     std::vector<std::vector<i32>> adj;
11     HLD(i32 n) : n(n), cur(0) {
12         info.resize(4 << std::lg(n));
13         tag.resize(4 << std::lg(n));
14         adj.assign(n, {});
15         fa.resize(n, -1);
16         siz.resize(n);
17         dep.resize(n);
18         top.resize(n);
19         val.resize(n);
20         id.resize(n);
21         w.resize(n);
22     }
23     void add(i32 p, const Tag &k) {
24         info[p].apply(k);
25         tag[p].apply(k);
26     }

```

```

27 void update(i32 p, const Tag &k) {
28     info[p].apply(k);
29     tag[p].apply(k);
30 }
31 void pull(i32 p) {
32     info[p] = info[p << 1] + info[p << 1 | 1];
33 }
34 void push(i32 p) {
35     update(p << 1, tag[p]);
36     update(p << 1 | 1, tag[p]);
37     tag[p] = Tag();
38 }
39 void addEdge(i32 u, i32 v) {
40     adj[u].push_back(v);
41     adj[v].push_back(u);
42 }
43 void assign(i32 u, i32 w) {
44     this->w[u] = w;
45 }
46 void work(i32 root = 0) {
47     top[root] = root;
48     dep[root] = 0;
49     fa[root] = -1;
50     dfs1(root);
51     dfs2(root);
52     auto build = [&](auto self, i32 p, i32 l, i32 r) {
53         if(r == l) {
54             info[p].init(val[l]);
55             return;
56         }
57         i32 m = (l + r) / 2;
58         self(self, p << 1, l, m);
59         self(self, p << 1 | 1, m + 1, r);
60         pull(p);
61     };
62     build(build, 1, 0, n - 1);
63 }
64 void dfs1(i32 u) {
65     if (fa[u] != -1) {
66         adj[u].erase(std::find(adj[u].begin(), adj[u].end(), fa[u]));
67     }
68     siz[u] = 1;
69     for (auto &v : adj[u]) {
70         dep[v] = dep[u] + 1;
71         fa[v] = u;
72         dfs1(v);
73         siz[u] += siz[v];
74         if (siz[adj[u][0]] < siz[v]) {

```

```

75         std::swap(v, adj[u][0]);
76     }
77 }
78 }
79 void dfs2(i32 u) {
80     id[u] = cur;
81     val[cur++] = w[u];
82     for (const auto &v : adj[u]) {
83         if (v == fa[u]) continue;
84         top[v] = v == adj[u][0] ? top[u] : v;
85         dfs2(v);
86     }
87 }
88 i32 lca(i32 u, i32 v) {
89     while (top[u] != top[v]) {
90         if (dep[top[u]] > dep[top[v]]) {
91             u = fa[top[u]];
92         } else {
93             v = fa[top[v]];
94         }
95     }
96     return dep[u] < dep[v] ? u : v;
97 }
98 void rangeApply(i32 p, i32 x, i32 y, i32 l, i32 r, const Tag &k) {
99     if (x <= l && y >= r) {
100         add(p, k);
101         return;
102     }
103     push(p);
104     i32 m = (l + r) / 2;
105     if (x <= m) rangeApply(p << 1, x, y, l, m, k);
106     if (y > m) rangeApply(p << 1 | 1, x, y, m + 1, r, k);
107     pull(p);
108 }
109 void pathApply(i32 u, i32 v, const Tag &k) {
110     while (top[u] != top[v]) {
111         if (dep[top[u]] < dep[top[v]]) {
112             std::swap(u, v);
113         }
114         rangeApply(1, id[top[u]], id[u], 0, n - 1, k);
115         u = fa[top[u]];
116     }
117     if (dep[u] < dep[v]) {
118         std::swap(u, v);
119     }
120     rangeApply(1, id[v], id[u], 0, n - 1, k);
121 }
122 void treeApply(i32 u, const Tag &k) {

```



```

123     rangeApply(1, id[u], id[u] + siz[u] - 1, 0, n - 1, k);
124 }
125 Info rangeQuery(i32 p, i32 x, i32 y, i32 l, i32 r) {
126     if (x <= l && y >= r) return info[p];
127     Info res{};
128     push(p);
129     i32 m = (l + r) / 2;
130     if (x <= m) res = (res + rangeQuery(p << 1, x, y, l, m));
131     if (y > m) res = (res + rangeQuery(p << 1 | 1, x, y, m + 1, r));
132     pull(p);
133     return res;
134 }
135 Info pathQuery(i32 u, i32 v) {
136     Info res{};
137     while (top[u] != top[v]) {
138         if (dep[top[u]] < dep[top[v]]) {
139             std::swap(u, v);
140         }
141         res = res + rangeQuery(1, id[top[u]], id[u], 0, n - 1);
142         u = fa[top[u]];
143     }
144     if (dep[u] < dep[v]) {
145         std::swap(u, v);
146     }
147     res = res + rangeQuery(1, id[v], id[u], 0, n - 1);
148     return res;
149 }
150 Info treeQuery(i32 u) {
151     return rangeQuery(1, id[u], id[u] + siz[u] - 1, 0, n - 1);
152 }
153 };
154
155 struct Tag {
156     i64 add = 0;
157     void apply(const Tag &v) {
158         add = add + v.add;
159     }
160 };
161
162 struct Info {
163     i64 sum = 0, len = 1;
164     i64 mn = INF, mx = -INF;
165     void init(const i64 &x) {
166         mn = mx = x;
167         sum = x;
168     }
169     void apply(const Tag &v) {
170         sum += len * v.add;

```

```

171         mn += v.add;
172         mx += v.add;
173     }
174     Info operator + (const Info &a) {
175         Info res;
176         res.mn = std::min(mn, a.mn);
177         res.mx = std::max(mx, a.mx);
178         res.sum = sum + a.sum;
179         res.len = len + a.len;
180         return res;
181     }
182 };

```

### 1.11.3 HLD

```

1 struct HLD {
2     i32 n, cur;
3     std::vector<i32> siz, top, dep, parent, in, out, seq;
4     std::vector<std::vector<i32>> adj;
5     HLD() {}
6     HLD(i32 n) {
7         init(n);
8     }
9     void init(i32 n) {
10         this->n = n;
11         in.resize(n);
12         out.resize(n);
13         siz.resize(n);
14         top.resize(n);
15         dep.resize(n);
16         seq.resize(n);
17         parent.resize(n);
18         adj.assign(n, {});
19         cur = 0;
20     }
21     void addEdge(i32 u, i32 v) {
22         adj[u].push_back(v);
23         adj[v].push_back(u);
24     }
25     void work(i32 root = 0) {
26         top[root] = root;
27         dep[root] = 0;
28         parent[root] = -1;
29         dfs1(root);
30         dfs2(root);
31     }
32     void dfs1(i32 u) {
33         if (parent[u] != -1) {

```

```

34     adj[u].erase(std::find(adj[u].begin(), adj[u].end(), parent[u]));
35 }
36 siz[u] = 1;
37 for (auto &v : adj[u]) {
38     parent[v] = u;
39     dep[v] = dep[u] + 1;
40     dfs1(v);
41     siz[u] += siz[v];
42     if (siz[v] > siz[adj[u][0]]) {
43         std::swap(v, adj[u][0]);
44     }
45 }
46 }
47 void dfs2(i32 u) {
48     in[u] = cur++;
49     seq[in[u]] = u;
50     for (auto v : adj[u]) {
51         top[v] = v == adj[u][0] ? top[u] : v;
52         dfs2(v);
53     }
54     out[u] = cur;
55 }
56 i32 lca(i32 u, i32 v) {
57     while (top[u] != top[v]) {
58         if (dep[top[u]] > dep[top[v]]) {
59             u = parent[top[u]];
60         } else {
61             v = parent[top[v]];
62         }
63     }
64     return dep[u] < dep[v] ? u : v;
65 }
66 i32 dist(i32 u, i32 v) {
67     return dep[u] + dep[v] - 2 * dep[lca(u, v)];
68 }
69 i32 jump(i32 u, i32 k) {
70     if (dep[u] < k) {
71         return -1;
72     }
73     i32 d = dep[u] - k;
74     while (dep[top[u]] > d) {
75         u = parent[top[u]];
76     }
77     return seq[in[u] - dep[u] + d];
78 }
79 bool isAncestor(i32 u, i32 v) {
80     return in[u] <= in[v] && in[v] < out[u];
81 }

```

```

82 i32 rootedParent(i32 u, i32 v) {
83     std::swap(u, v);
84     if (u == v) {
85         return u;
86     }
87     if (!isAncestor(u, v)) {
88         return parent[u];
89     }
90     auto dfnCmp = [&](const i32 &x, const i32 &y) {
91         return in[x] < in[y];
92     };
93     auto it = std::upper_bound(adj[u].begin(), adj[u].end(), v, dfnCmp) -
94     1;
95     return *it;
96 }
97 i32 rootedSize(i32 u, i32 v) {
98     if (u == v) {
99         return n;
100     }
101     if (!isAncestor(v, u)) {
102         return siz[v];
103     }
104     return n - siz[rootedParent(u, v)];
105 }
106 i32 rootedLca(i32 a, i32 b, i32 c) {
107     return lca(a, b) ^ lca(b, c) ^ lca(c, a);
108 }

```

## 1.12 环计数

### 1.12.1 CommonRing

```

1 i64 countRings(const std::vector<std::vector<i32>> &adj) {
2     i32 n = adj.size();
3     std::vector dp(1 << n, std::vector<i64>(n));
4     for (i32 i = 0; i < n; ++i) {
5         dp[1 << i][i] = 1;
6     }
7     i64 cnt = 0;
8     for (i32 i = 0; i < n; ++i) {
9         cnt += adj[i].size();
10    }
11    i64 res = 0;
12    for (i32 s = 1; s < (1 << n); ++s) {
13        for (i32 u = 0; u < n; ++u) {
14            if (!dp[s][u]) continue;
15            for (const auto &v : adj[u]) {

```

```

16         if ((s & -s) > (1 << v)) {
17             continue;
18         }
19         if (s & (1 << v)) {
20             if ((s & -s) == (1 << v)) {
21                 res += dp[s][u];
22             }
23         } else {
24             dp[s | (1 << v)][v] += dp[s][u];
25         }
26     }
27 }
28 }
29 res = (res - cnt / 2) / 2;
30 return res;
31 }

```

### 1.12.2 QuarternaryRing

```

1 i64 countRings(const std::vector<std::vector<i32>> &adj) {
2     i32 n = adj.size();
3     std::vector<i32> deg(n);
4     for (i32 i = 0; i < n; ++i) {
5         deg[i] = adj[i].size();
6     }
7     std::vector<std::vector<i32>> nadj(n);
8     for (i32 u = 0; u < n; ++u) {
9         for (const auto &v : adj[u]) {
10             if (deg[u] == deg[v] && u > v) {
11                 nadj[u].push_back(v);
12             }
13             if (deg[u] > deg[v]) {
14                 nadj[u].push_back(v);
15             }
16         }
17     }
18     i64 res = 0;
19     std::vector<i32> cnt(n);
20     for (i32 u = 0; u < n; ++u) {
21         for (const auto &v : nadj[u]) {
22             for (const auto &w : adj[v]) {
23                 if (deg[u] == deg[w] && u <= w) continue;
24                 if (deg[u] < deg[w]) continue;
25                 res += cnt[w];
26                 cnt[w]++;
27             }
28         }
29         for (const auto &v : nadj[u]) {

```

```

30             for (const auto &w : adj[v]) {
31                 cnt[w] = 0;
32             }
33         }
34     }
35     return res;
36 }

```

### 1.12.3 TernaryRing

```

1 i64 countRings(const std::vector<std::vector<i32>> &adj) {
2     i32 n = adj.size();
3     std::vector<i32> deg(n);
4     for (i32 u = 0; u < n; ++u) {
5         for (const auto &v : adj[u]) {
6             deg[v]++;
7         }
8     }
9     std::vector<std::vector<i32>> nadj(n);
10    for (i32 u = 0; u < n; ++u) {
11        for (const auto &v : adj[u]) {
12            if (deg[u] == deg[v] && u < v) {
13                nadj[u].push_back(v);
14            }
15            if (deg[u] < deg[v]) {
16                nadj[u].push_back(v);
17            }
18        }
19    }
20    i64 res = 0;
21    std::vector<i32> time(n, -1);
22    for (i32 u = 0; u < n; ++u) {
23        for (const auto &v : nadj[u]) {
24            time[v] = u;
25        }
26        for (const auto &v : nadj[u]) {
27            for (const auto &w : nadj[v]) {
28                if (time[w] == u) {
29                    res++;
30                }
31            }
32        }
33    }
34    return res;
35 }

```

## 1.13 生成树

### 1.13.1 DMST

```
1 const i64 INF = 1E18;
2
3 struct DSU {
4     std::vector<i32> p, siz;
5     DSU() {}
6     DSU(i32 n) {
7         init(n);
8     }
9     void init(i32 n) {
10         p.resize(n);
11         siz.assign(n, 1);
12         std::iota(p.begin(), p.end(), 0);
13     }
14     i32 find(i32 x) {
15         while (x != p[x]) x = p[x] = p[p[x]];
16         return x;
17     }
18     bool same(i32 x, i32 y) {
19         return find(x) == find(y);
20     }
21     bool merge(i32 x, i32 y) {
22         x = find(x);
23         y = find(y);
24         if (x == y) {
25             return false;
26         } else {
27             p[y] = x;
28             siz[x] += siz[y];
29             return true;
30         }
31     }
32     i32 size(i32 x) {
33         return siz[find(x)];
34     }
35 };
36
37 template <typename Info, typename Tag, typename Func = std::less<Info>>
38 struct LeftistTree {
39     struct Node {
40         Info info;
41         Tag tag;
42         i32 dis = 0;
43         Node *lc = nullptr;
44         Node *rc = nullptr;
45         Node(const Info &x) : info(x) {}
```

```
46     } *root = nullptr;
47     i32 siz = 0;
48     Func cmp;
49     LeftistTree() = default;
50     LeftistTree(Func cmp) : cmp(cmp) {}
51     ~LeftistTree() {
52         clear();
53     }
54     void abdicate() {
55         root = nullptr;
56         siz = 0;
57     }
58     i32 dist(Node *x) {
59         if (x == nullptr) {
60             return -1;
61         }
62         return x->dis;
63     }
64     void pushdown(Node *x) {
65         if (x == nullptr) {
66             return;
67         }
68         x->info.apply(x->tag);
69         if (x->lc != nullptr) {
70             x->lc->tag.apply(x->tag);
71         }
72         if (x->rc != nullptr) {
73             x->rc->tag.apply(x->tag);
74         }
75         x->tag = Tag{};
76     }
77     Node* merge(Node* x, Node* y) {
78         if (x == nullptr) return y;
79         if (y == nullptr) return x;
80         pushdown(x);
81         pushdown(y);
82         if (cmp(x->info, y->info)) {
83             std::swap(x, y);
84         }
85         x->rc = merge(x->rc, y);
86         if (dist(x->lc) < dist(x->rc)) {
87             std::swap(x->lc, x->rc);
88         }
89         x->dis = dist(x->rc) + 1;
90         return x;
91     }
92     void merge(LeftistTree &x) {
93         root = merge(root, x.root);
```

```

94     siz += x.size();
95     x.abdicate();
96 }
97 void clear() {
98     if (root == nullptr) return;
99     std::queue<Node*> q;
100    q.push(root);
101    while (!q.empty()) {
102        Node *u = q.front();
103        q.pop();
104        if (u->lc != nullptr) {
105            q.push(u->lc);
106        }
107        if (u->rc != nullptr) {
108            q.push(u->rc);
109        }
110        delete u;
111    }
112    abdicate();
113 }
114 void push(Info x) {
115     Node *temp = new Node(x);
116     root = merge(root, temp);
117     siz++;
118 }
119 void pop() {
120     pushdown(root);
121     Node *temp = root;
122     root = merge(root->lc, root->rc);
123     delete temp;
124     siz--;
125 }
126 void apply(const Tag &v) {
127     if (root != nullptr) {
128         root->tag.apply(v);
129     }
130 }
131 bool empty() {
132     return root == nullptr;
133 }
134 Info top() {
135     pushdown(root);
136     return root->info;
137 }
138 i32 size() {
139     return siz;
140 }
141 };

```

```

142
143 struct Tag {
144     i64 add = 0;
145     void apply(const Tag &v) {
146         add += v.add;
147     }
148 };
149
150 struct Info {
151     i32 u, v;
152     i64 w;
153     void apply(const Tag &v) {
154         w += v.add;
155     }
156     bool operator < (const Info &u) const {
157         return w < u.w;
158     }
159     bool operator > (const Info &u) const {
160         return w > u.w;
161     }
162 };
163
164 i64 dmst(i32 n, i32 r, const std::vector<Info> &edg) {
165     i32 m = edg.size();
166     std::vector<vector<Info>> lt(2 * n, LeftistTree<Info, Tag, std::greater<Info>>(std::greater<Info>()));
167     for (const auto &e : edg) {
168         lt[e.v].push(e);
169     }
170     for (i32 i = 0; i < n; ++i) {
171         i32 u = i;
172         i32 v = (i + 1) % n;
173         lt[v].push({u, v, INF});
174     }
175     DSU dsu(2 * n);
176     std::vector<i32> stk;
177     std::vector<bool> vis(2 * n);
178     stk.push_back(r);
179     vis[r] = true;
180     i64 res = 0;
181     i32 cur = n;
182     while (!lt[stk.back()].empty()) {
183         i32 u = stk.back();
184         i32 rt = dsu.find(lt[u].top().u);
185         if (rt == u) {
186             lt[u].pop();
187             continue;
188         }

```

```

189     if (!vis[rt]) {
190         stk.push_back(rt);
191         vis[rt] = true;
192         continue;
193     }
194     i32 nu = cur++;
195     while (vis[rt]) {
196         i32 v = stk.back();
197         stk.pop_back();
198         vis[v] = false;
199         dsu.merge(nu, v);
200         auto mn = lt[v].top();
201         lt[v].pop();
202         lt[v].apply({-mn.w});
203         lt[nu].merge(lt[v]);
204         if (!dsu.same(mn.v, r)) {
205             res += mn.w;
206         }
207     }
208     stk.push_back(nu);
209     vis[nu] = true;
210 }
211 return res >= INF ? -1 : res;
212 }

```

### 1.13.2 Kruskal

```

1 struct DSU {
2     std::vector<i32> p, siz;
3     DSU() {}
4     DSU(i32 n) {
5         init(n);
6     }
7     void init(i32 n) {
8         p.resize(n);
9         siz.assign(n, 1);
10        std::iota(p.begin(), p.end(), 0);
11    }
12    i32 find(i32 x) {
13        while (x != p[x]) x = p[x] = p[p[x]];
14        return x;
15    }
16    bool same(i32 x, i32 y) {
17        return find(x) == find(y);
18    }
19    bool merge(i32 x, i32 y) {
20        x = find(x);
21        y = find(y);

```

```

22     if (x == y) {
23         return false;
24     } else {
25         p[y] = x;
26         siz[x] += siz[y];
27         return true;
28     }
29 }
30 i32 size(i32 x) {
31     return siz[find(x)];
32 }
33 };
34
35 struct Node {
36     i32 u, v;
37     i64 w;
38     bool operator < (const Node &u) const {
39         return w > u.w;
40     }
41 };
42
43 i64 kruskal(i32 n, std::vector<Node> &edg) {
44     std::priority_queue<Node> pq;
45     for (const auto &e : edg) {
46         pq.push(e);
47     }
48     i64 res = 0;
49     DSU dsu(n);
50     while (!pq.empty()) {
51         auto [u, v, w] = pq.top();
52         pq.pop();
53         if (dsu.merge(u, v)) {
54             res += w;
55         }
56     }
57     if (dsu.size(0) != n) {
58         return -1;
59     }
60     return res;
61 }

```

### 1.13.3 SSST

```

1 const i64 INF = 1E18;
2 const i64 MOD = 1E9 + 7;
3
4 struct Node {
5     i32 u, v;

```

```

6   i64 w;
7   bool operator < (const Node &u) const {
8       return w < u.w;
9   }
10 };
11
12 template <typename Info, typename Tag>
13 struct HLD {
14     i32 n, cur;
15     std::vector<Tag> tag;
16     std::vector<Info> info;
17     std::vector<i32> w, id, fa, val, siz, dep, top;
18     std::vector<std::vector<i32>> adj;
19     std::vector<Node> edg;
20     HLD(i32 n) : n(n), cur(0) {
21         info.resize(4 << std::__lg(n));
22         tag.resize(4 << std::__lg(n));
23         adj.assign(n, {});
24         fa.resize(n, -1);
25         val.resize(n);
26         siz.resize(n);
27         dep.resize(n);
28         top.resize(n);
29         id.resize(n);
30         w.resize(n);
31     }
32     void add(i32 p, const Tag &k) {
33         info[p].apply(k);
34         tag[p].apply(k);
35     }
36     void update(i32 p, const Tag &k) {
37         info[p].apply(k);
38         tag[p].apply(k);
39     }
40     void pull(i32 p) {
41         info[p] = info[p << 1] + info[p << 1 | 1];
42     }
43     void push(i32 p) {
44         update(p << 1, tag[p]);
45         update(p << 1 | 1, tag[p]);
46         tag[p] = Tag();
47     }
48     void addEdge(i32 u, i32 v, i32 w) {
49         adj[u].push_back(v);
50         adj[v].push_back(u);
51         edg.push_back({u, v, w});
52     }
53     void work(i32 root = 0) {

```

```

54     top[root] = root;
55     dep[root] = 0;
56     fa[root] = -1;
57     dfs1(root);
58     dfs2(root);
59     for (auto &[u, v, w] : edg) {
60         i64 p = lca(u, v);
61         if (v == p) {
62             val[id[u]] = w;
63         } else {
64             val[id[v]] = w;
65         }
66     }
67     auto build = [&](auto self, i32 p, i32 l, i32 r) {
68         if (r == l) {
69             info[p].init(val[l]);
70             return;
71         }
72         i32 m = (l + r) / 2;
73         self(self, p << 1, l, m);
74         self(self, p << 1 | 1, m + 1, r);
75         pull(p);
76     };
77     build(build, 1, 0, n - 1);
78 }
79 void dfs1(i32 u) {
80     if (fa[u] != -1) {
81         adj[u].erase(std::find(adj[u].begin(), adj[u].end(), fa[u]));
82     }
83     siz[u] = 1;
84     for (auto &v : adj[u]) {
85         dep[v] = dep[u] + 1;
86         fa[v] = u;
87         dfs1(v);
88         siz[u] += siz[v];
89         if (siz[adj[u][0]] < siz[v]) {
90             std::swap(v, adj[u][0]);
91         }
92     }
93 }
94 void dfs2(i32 u) {
95     id[u] = cur;
96     val[cur++] = w[u];
97     for (const auto &v : adj[u]) {
98         if (v == fa[u]) continue;
99         top[v] = v == adj[u][0] ? top[u] : v;
100         dfs2(v);
101     }

```

```

102 }
103 i32 lca(i32 u, i32 v) {
104     while (top[u] != top[v]) {
105         if (dep[top[u]] > dep[top[v]]) {
106             u = fa[top[u]];
107         } else {
108             v = fa[top[v]];
109         }
110     }
111     return dep[u] < dep[v] ? u : v;
112 }
113 void rangeApply(i32 p, i32 x, i32 y, i32 l, i32 r, const Tag &k) {
114     if (x <= l && y >= r) {
115         add(p, k);
116         return;
117     }
118     push(p);
119     i32 m = (l + r) / 2;
120     if (x <= m) rangeApply(p << 1, x, y, l, m, k);
121     if (y > m) rangeApply(p << 1 | 1, x, y, m + 1, r, k);
122     pull(p);
123 }
124 void pathApply(i32 u, i32 v, const Tag &k) {
125     while (top[u] != top[v]) {
126         if (dep[top[u]] < dep[top[v]]) {
127             std::swap(u, v);
128         }
129         rangeApply(1, id[top[u]], id[u], 0, n - 1, k);
130         u = fa[top[u]];
131     }
132     if (dep[u] < dep[v]) {
133         std::swap(u, v);
134     }
135     if (u != v) {
136         rangeApply(1, id[v] + 1, id[u], 0, n - 1, k);
137     }
138 }
139 void treeApply(i32 u, const Tag &k) {
140     rangeApply(1, id[u], id[u] + siz[u] - 1, 0, n - 1, k);
141 }
142 Info rangeQuery(i32 p, i32 x, i32 y, i32 l, i32 r) {
143     if (x <= l && y >= r) return info[p];
144     Info res{};
145     push(p);
146     i32 m = (l + r) / 2;
147     if (x <= m) res = (res + rangeQuery(p << 1, x, y, l, m));
148     if (y > m) res = (res + rangeQuery(p << 1 | 1, x, y, m + 1, r));
149     pull(p);

```

```

150     return res;
151 }
152 Info pathQuery(i32 u, i32 v) {
153     Info res{};
154     while (top[u] != top[v]) {
155         if (dep[top[u]] < dep[top[v]]) {
156             std::swap(u, v);
157         }
158         res = res + rangeQuery(1, id[top[u]], id[u], 0, n - 1);
159         u = fa[top[u]];
160     }
161     if (dep[u] < dep[v]) {
162         std::swap(u, v);
163     }
164     if (u != v) {
165         res = res + rangeQuery(1, id[v] + 1, id[u], 0, n - 1);
166     }
167     return res;
168 }
169 Info treeQuery(i32 u) {
170     return rangeQuery(1, id[u], id[u] + siz[u] - 1, 0, n - 1);
171 }
172 };
173
174 struct Tag {
175     i64 add = 0;
176     void apply(const Tag &v) {
177         add = add + v.add;
178     }
179 };
180
181 struct Info {
182     i64 sum = 0, len = 1;
183     std::array<i64, 2> mn{INF, INF};
184     std::array<i64, 2> mx{-INF, -INF};
185     void init(const i64 &x) {
186         mn[0] = mx[0] = x;
187         sum = x;
188     }
189     void apply(const Tag &v) {
190         sum += len * v.add;
191         mn[0] += v.add;
192         mx[0] += v.add;
193         mn[1] += v.add;
194         mx[1] += v.add;
195     }
196     Info operator + (const Info &a) {
197         Info res;

```



```

198     if (mn[0] < a.mn[0]) {
199         res.mn[0] = mn[0];
200         res.mn[1] = std::min({mn[1], a.mn[0], a.mn[1]});
201     } else if (mn[0] > a.mn[0]) {
202         res.mn[0] = a.mn[0];
203         res.mn[1] = std::min({mn[0], mn[1], a.mn[1]});
204     } else {
205         res.mn[0] = mn[0];
206         res.mn[1] = std::min(mn[1], a.mn[1]);
207     }
208     if (mx[0] > a.mx[0]) {
209         res.mx[0] = mx[0];
210         res.mx[1] = std::max({mx[1], a.mx[0], a.mx[1]});
211     } else if (mx[0] < a.mx[0]) {
212         res.mx[0] = a.mx[0];
213         res.mx[1] = std::max({mx[0], mx[1], a.mx[1]});
214     } else {
215         res.mx[0] = mx[0];
216         res.mx[1] = std::max(mx[1], res.mx[1]);
217     }
218     res.sum = sum + a.sum;
219     res.len = len + a.len;
220     return res;
221 }
222 };
223
224 struct DSU {
225     std::vector<i32> p, siz;
226     DSU() {}
227     DSU(i32 n) {
228         init(n);
229     }
230     void init(i32 n) {
231         p.resize(n);
232         siz.assign(n, 1);
233         std::iota(p.begin(), p.end(), 0);
234     }
235     i32 find(i32 x) {
236         while (x != p[x]) x = p[x] = p[p[x]];
237         return x;
238     }
239     bool same(i32 x, i32 y) {
240         return find(x) == find(y);
241     }
242     bool merge(i32 x, i32 y) {
243         x = find(x);
244         y = find(y);
245         if (x == y) {

```

```

246             return false;
247         } else {
248             p[y] = x;
249             siz[x] += siz[y];
250             return true;
251         }
252     }
253     i32 size(i32 x) {
254         return siz[find(x)];
255     }
256 };
257
258 i64 ssst(i32 n, std::vector<Node> &edg) {
259     std::vector<std::vector<i32>> adj(n);
260     std::sort(edg.begin(), edg.end());
261     i32 m = edg.size();
262     i64 res = 0;
263     DSU dsu(n);
264     HLD<Info, Tag> hld(n);
265     std::vector<bool> vis(m);
266     for (i32 i = 0; i < m; ++i) {
267         auto [u, v, w] = edg[i];
268         if (dsu.merge(u, v)) {
269             hld.addEdge(u, v, w);
270             vis[i] = true;
271             res += w;
272         }
273     }
274     hld.work();
275     i64 fix = INF;
276     for (i32 i = 0; i < m; ++i) {
277         if (vis[i]) {
278             continue;
279         }
280         auto [u, v, w] = edg[i];
281         if (u == v) {
282             continue;
283         }
284         auto cur = hld.pathQuery(u, v);
285         if (w > cur.mx[0]) {
286             fix = std::min(fix, w - cur.mx[0]);
287         } else if (w > cur.mx[1]) {
288             fix = std::min(fix, w - cur.mx[1]);
289         }
290     }
291     res += fix;
292     return res;
293 }

```

### 1.13.4 SteinerTree

```
1 const i64 INF = 1E18;
2
3 struct Node {
4     i32 v, w;
5 };
6
7 std::vector<bool> vis;
8 std::vector<std::vector<i64>> dp;
9
10 void spfa(i32 s, std::vector<std::vector<Node>> &adj) {
11     i32 n = adj.size();
12     vis.assign(n, false);
13     std::queue<i32> q;
14     for (i32 i = 0; i < n; ++i) {
15         if (dp[s][i] != INF) {
16             vis[i] = true;
17             q.push(i);
18         }
19     }
20     while (!q.empty()) {
21         i32 u = q.front();
22         q.pop();
23         vis[u] = false;
24         for (const auto &[v, w] : adj[u]) {
25             if (dp[s][u] + w < dp[s][v]) {
26                 dp[s][v] = dp[s][u] + w;
27                 if (!vis[v]) {
28                     vis[v] = true;
29                     q.push(v);
30                 }
31             }
32         }
33     }
34 }
35
36 i64 steiner(std::vector<std::vector<Node>> &adj, std::vector<i32> &num) {
37     i32 n = adj.size();
38     i32 m = num.size();
39     dp.assign(1 << m, std::vector<i64>(n, INF));
40     for (i32 i = 0; i < m; ++i) {
41         dp[1 << i][num[i]] = 0;
42     }
43     for (i32 s = 1; s < (1 << m); ++s) {
44         for (i32 t = (s - 1) & s; t > 0; t = (t - 1) & s) {
45             if (t < (s ^ t)) {
46                 break;
47             }
48         }
49     }
```

```
48         for (i32 i = 0; i < n; ++i) {
49             dp[s][i] = std::min(dp[s][i], dp[t][i] + dp[s ^ t][i]);
50         }
51     }
52     spfa(s, adj);
53 }
54 return *std::min_element(dp[(1 << m) - 1].begin(), dp[(1 << m) - 1].end());
55 }
```

## 1.14 矩阵树定理

### 1.14.1 MatrixTree-Directed

```
1 const i64 MOD = 1E9 + 7;
2
3 template <typename T>
4 struct Matrix {
5     i32 n, m;
6     std::vector<std::vector<T>> v;
7     Matrix(i32 n, i32 m) : n(n), m(m) {
8         v = std::vector<std::vector<T>>(n, std::vector<T>(m));
9     }
10    Matrix(const std::vector<std::vector<T>> &v) : Matrix(v.size(), v[0].size())
11    {
12        for (i32 i = 0; i < n; ++i) {
13            for (i32 j = 0; j < m; ++j) {
14                this->v[i][j] = v[i][j];
15            }
16        }
17    }
18    std::vector<T> &operator [] (i32 x) {
19        assert(x < n);
20        return v[x];
21    }
22    Matrix<T> operator = (const Matrix<T> &x) {
23        n = x.n;
24        m = x.m;
25        v = x;
26        return *this;
27    }
28    Matrix<T> operator + (const Matrix<T> &x) {
29        assert(n == x.n && m == x.m);
30        Matrix<T> res(n, m);
31        for (i32 i = 0; i < n; ++i) {
32            for (i32 j = 0; j < m; ++j) {
33                res[i][j] = v[i][j] + x[i][j];
34            }
35        }
```

```

35     return res;
36 }
37 Matrix<T> operator += (const Matrix<T> &x) {
38     return *this = *this + x;
39 }
40 Matrix<T> operator - (const Matrix<T> &x) {
41     assert(n == x.n && m == x.m);
42     Matrix<T> res(n, m);
43     for (i32 i = 0; i < n; ++i) {
44         for (i32 j = 0; j < m; ++j) {
45             res[i][j] = v[i][j] - v[i][j];
46         }
47     }
48     return res;
49 }
50 Matrix<T> operator -= (const Matrix<T> &x) {
51     return *this = *this - x;
52 }
53 Matrix<T> operator * (const Matrix<T> &x) {
54     assert(m == x.n);
55     Matrix<T> res(n, x.m);
56     for (i32 i = 0; i < n; ++i) {
57         for (i32 j = 0; j < x.m; ++j) {
58             for (i32 k = 0; k < m; ++k) {
59                 res[i][j] = res[i][j] + v[i][k] * v[k][j];
60             }
61         }
62     }
63     return res;
64 }
65 Matrix<T> operator *= (const Matrix<T> &x) {
66     return *this = *this * x;
67 }
68 static Matrix<T> power(Matrix<T> a, i64 b) {
69     Matrix<T> res(Matrix::eye(a.n));
70     while (b > 0) {
71         if (b & 1) res = res * a;
72         a = a * a;
73         b >>= 1;
74     }
75     return res;
76 }
77 static Matrix<T> eye(i32 n) {
78     Matrix<T> res(n, n);
79     for (i32 i = 0; i < n; ++i) {
80         res[i][i] = 1;
81     }
82     return res;

```

```

83 }
84 friend std::ostream& operator << (std::ostream &os, const Matrix<T> &x) {
85     for (i32 i = 0; i < x.n; ++i) {
86         for (i32 j = 0; j < x.m; ++j) {
87             os << x.v[i][j] << " \n"[j + 1 == x.m];
88         }
89     }
90     return os;
91 }
92 };
93
94 template <typename T>
95 i64 det(Matrix<T> x) {
96     assert(x.n == x.m);
97     i64 res = 1;
98     for (i32 i = 0; i < x.n; ++i) {
99         i32 pivot = i;
100         for (i32 j = i; j < x.n; ++j) {
101             if (x[j][i] != 0) {
102                 pivot = j;
103                 break;
104             }
105         }
106         if (x[pivot][i] == 0) {
107             return 0;
108         }
109         if (pivot != i) {
110             std::swap(x[i], x[pivot]);
111             res = (MOD - res) % MOD;
112         }
113         for (i32 j = i + 1; j < x.n; ++j) {
114             if (x[j][i] == 0) continue;
115             while (x[i][i] != 0) {
116                 i64 fix = x[j][i] / x[i][i];
117                 for (i32 k = i; k < x.n; ++k) {
118                     x[j][k] = ((x[j][k] - fix * x[i][k]) % MOD + MOD) % MOD;
119                 }
120                 for (i32 k = i; k < x.n; ++k) {
121                     std::swap(x[i][k], x[j][k]);
122                 }
123                 res = (MOD - res) % MOD;
124             }
125             for (i32 k = i; k < x.n; ++k) {
126                 std::swap(x[i][k], x[j][k]);
127             }
128             res = (MOD - res) % MOD;
129         }
130         res = res * x[i][i] % MOD;

```

```

131     }
132     return res;
133 }
134
135 template <typename T>
136 struct MatrixTree {
137     i32 n;
138     Matrix<T> root, leaf;
139     MatrixTree(const Matrix<T> &adj) : n(adj.n), root(n, n), leaf(n, n) {
140         assert(adj.n == adj.m);
141         for (i32 i = 0; i < n; ++i) {
142             for (i32 j = 0; j < n; ++j) {
143                 root[i][i] += adj[i][j];
144                 leaf[i][i] += adj[j][i];
145                 root[i][j] -= adj[i][j];
146                 leaf[i][j] -= adj[j][i];
147             }
148         }
149     }
150     MatrixTree(const std::vector<std::vector<i32>> &adj) : n(adj.size()), root(
151         n, n), leaf(n, n) {
152         for (i32 i = 0; i < n; ++i) {
153             root[i][i] = adj.size();
154             for (const auto &j : adj[i]) {
155                 leaf[j][j]++;
156                 leaf[j][i]--;
157                 root[i][j]--;
158             }
159         }
160         i64 cntRoot(i32 rt = 0) {
161             i32 x = 0, y = 0;
162             Matrix<T> sub(n - 1, n - 1);
163             for (i32 i = 0; i < n; ++i) {
164                 if (i == rt) continue;
165                 for (i32 j = 0; j < n; ++j) {
166                     if (j == rt) continue;
167                     sub[x][y] = out[i][j];
168                     y = (y + 1) % (n - 1);
169                 }
170                 x++;
171             }
172             return det(sub);
173         }
174         i64 cntLeaf(i32 rt = 0) {
175             i32 x = 0, y = 0;
176             Matrix<T> sub(n - 1, n - 1);
177             for (i32 i = 0; i < n; ++i) {

```

```

178                 if (i == rt) continue;
179                 for (i32 j = 0; j < n; ++j) {
180                     if (j == rt) continue;
181                     sub[x][y] = in[i][j];
182                     y = (y + 1) % (n - 1);
183                 }
184                 x++;
185             }
186             return det(sub);
187         }
188     };

```

### 1.14.2 MatrixTree-Undirected

```

1  const i64 MOD = 1E9 + 7;
2
3  i64 power(i64 a, i64 b, i64 p = MOD) {
4      i64 res = 1;
5      while (b) {
6          if (b & 1) res = res * a % p;
7          a = a * a % p;
8          b >>= 1;
9      }
10     return res;
11 }
12
13 i64 inv(i64 a, i64 p = MOD) {
14     return power(a, p - 2, p);
15 }
16
17 template <typename T>
18 struct Matrix {
19     i32 n, m;
20     std::vector<std::vector<T>> v;
21     Matrix(i32 n, i32 m) : n(n), m(m) {
22         v = std::vector<std::vector<T>>(m);
23     }
24     Matrix(const std::vector<std::vector<T>> &v) : Matrix(v.size(), v[0].size()
25         ) {
26         for (i32 i = 0; i < n; ++i) {
27             for (i32 j = 0; j < m; ++j) {
28                 this->v[i][j] = v[i][j];
29             }
30         }
31     }
32     std::vector<T>& operator [] (i32 x) {
33         assert(x < n);
34         return v[x];

```

```

34 }
35 Matrix<T> operator = (const Matrix<T> &x) {
36     n = x.n;
37     m = x.m;
38     v = x;
39     return *this;
40 }
41 Matrix<T> operator + (const Matrix<T> &x) {
42     assert(n == x.n && m == x.m);
43     Matrix<T> res(n, m);
44     for (i32 i = 0; i < n; ++i) {
45         for (i32 j = 0; j < m; ++j) {
46             res[i][j] = v[i][j] + v[i][j];
47         }
48     }
49     return res;
50 }
51 Matrix<T> operator += (const Matrix<T> &x) {
52     return *this = *this + x;
53 }
54 Matrix<T> operator - (const Matrix<T> &x) {
55     assert(n == x.n && m == x.m);
56     Matrix<T> res(n, m);
57     for (i32 i = 0; i < n; ++i) {
58         for (i32 j = 0; j < m; ++j) {
59             res[i][j] = v[i][j] - v[i][j];
60         }
61     }
62     return res;
63 }
64 Matrix<T> operator -= (const Matrix<T> &x) {
65     return *this = *this - x;
66 }
67 Matrix<T> operator * (const Matrix<T> &x) {
68     assert(m == x.n);
69     Matrix<T> res(n, x.m);
70     for (i32 i = 0; i < n; ++i) {
71         for (i32 j = 0; j < x.m; ++j) {
72             for (i32 k = 0; k < m; ++k) {
73                 res[i][j] = res[i][j] + v[i][k] * v[k][j];
74             }
75         }
76     }
77     return res;
78 }
79 Matrix<T> operator *= (const Matrix<T> &x) {
80     return *this = *this * x;
81 }

```

```

82 static Matrix<T> power(Matrix<T> a, i64 b) {
83     Matrix<T> res(Matrix::eye(a.n));
84     while (b > 0) {
85         if (b & 1) res = res * a;
86         a = a * a;
87         b >>= 1;
88     }
89     return res;
90 }
91 static Matrix<T> eye(i32 n) {
92     Matrix<T> res(n, n);
93     for (i32 i = 0; i < n; ++i) {
94         res[i][i] = 1;
95     }
96     return res;
97 }
98 friend std::ostream& operator << (std::ostream &s, const Matrix<T> &x) {
99     for (i32 i = 0; i < x.n; ++i) {
100         for (i32 j = 0; j < x.m; ++j) {
101             os << x.v[i][j] << " \n"[j + 1 == x.m];
102         }
103     }
104     return os;
105 }
106 };
107
108 template <typename T>
109 i64 det(Matrix<T> x) {
110     assert(x.n == x.m);
111     i64 res = 1;
112     for (i32 i = 0; i < x.n; ++i) {
113         i32 pivot = i;
114         for (i32 j = i; j < x.n; ++j) {
115             if (x[j][i] != 0) {
116                 pivot = j;
117                 break;
118             }
119         }
120         if (x[pivot][i] == 0) {
121             return 0;
122         }
123         if (pivot != i) {
124             std::swap(x[i], x[pivot]);
125             res = (MOD - res) % MOD;
126         }
127         for (i32 j = i + 1; j < x.n; ++j) {
128             if (x[j][i] == 0) continue;
129             while (x[i][i] != 0) {

```

```

130         i64 fix = x[j][i] / x[i][i];
131         for (i32 k = i; k < x.n; ++k) {
132             x[j][k] = ((x[j][k] - fix * x[i][k]) % MOD + MOD) % MOD;
133         }
134         for (i32 k = i; k < x.n; ++k) {
135             std::swap(x[i][k], x[j][k]);
136         }
137         res = (MOD - res) % MOD;
138     }
139     for (i32 k = i; k < x.n; ++k) {
140         std::swap(x[i][k], x[j][k]);
141     }
142     res = (MOD - res) % MOD;
143 }
144 res = res * x[i][i] % MOD;
145 }
146 return res;
147 }
148
149 template <typename T>
150 struct MatrixTree {
151     i32 n;
152     Matrix<T> laplace;
153     MatrixTree(const Matrix<T> &adj) : n(adj.n), laplace(n, n) {
154         assert(adj.n == adj.m);
155         for (i32 i = 0; i < n; ++i) {
156             for (i32 j = 0; j < n; ++j) {
157                 laplace[i][i] += adj[i][j];
158                 laplace[i][j] -= adj[i][j];
159             }
160         }
161     }
162     MatrixTree(const std::vector<std::vector<T>> &adj) : n(adj.size()), laplace
(n, n) {
163         for (i32 i = 0; i < n; ++i) {
164             laplace[i][i] = adj[i].size();
165             for (const auto &j : adj[i]) {
166                 laplace[i][j]--;
167             }
168         }
169     }
170     i64 cntTree() {
171         Matrix<T> sub(n - 1, n - 1);
172         for (i32 i = 0; i < n - 1; ++i) {
173             for (i32 j = 0; j < n - 1; ++j) {
174                 sub[i][j] = laplace[i][j];
175             }
176         }

```

```

177         return det(sub);
178     }
179 };

```

## 1.15 缩点

### 1.15.1 EBCC

```

1  std::set<std::pair<i32, i32>> edg;
2
3  struct EBCC {
4      i32 n;
5      std::vector<std::vector<std::pair<i32, i32>>> adj;
6      std::vector<i32> dfn, low, bel;
7      std::vector<i32> stk;
8      i32 cur, cnt, tot;
9      EBCC() {}
10     EBCC(i32 n) {
11         init(n);
12     }
13     void init(i32 n) {
14         this->n = n;
15         cur = cnt = tot = 0;
16         adj.assign(n, {});
17         dfn.assign(n, -1);
18         bel.assign(n, -1);
19         low.resize(n);
20         stk.clear();
21     }
22     void addEdge(i32 u, i32 v) {
23         if (u == v) return;
24         adj[u].push_back({v, tot});
25         adj[v].push_back({u, tot});
26         tot++;
27     }
28     void dfs(i32 lst, i32 u) {
29         dfn[u] = low[u] = cur++;
30         stk.push_back(u);
31         for (auto &[v, nxt] : adj[u]) {
32             if (lst == nxt) continue;
33             if (dfn[v] == -1) {
34                 edg.emplace(u, v);
35                 dfs(u, v);
36                 low[u] = std::min(low[u], low[v]);
37             } else if (bel[v] == -1 && dfn[v] < dfn[u]) {
38                 edg.emplace(u, v);
39                 low[u] = std::min(low[u], dfn[v]);
40             }

```

```

41     }
42     if (dfn[u] == low[u]) {
43         i32 v;
44         do {
45             v = stk.back();
46             bel[v] = cnt;
47             stk.pop_back();
48         } while (v != u);
49         cnt++;
50     }
51 }
52 std::vector<i32> work() {
53     dfs(-1, 0);
54     return bel;
55 }
56 struct Graph {
57     i32 n;
58     std::vector<std::pair<i32, i32>> edges;
59     std::vector<i32> siz;
60     std::vector<i32> cnte;
61 };
62 Graph compress() {
63     Graph g;
64     g.n = cnt;
65     g.siz.resize(cnt);
66     g.cnte.resize(cnt);
67     for (i32 i = 0; i < n; ++i) {
68         g.siz[bel[i]]++;
69         for (auto &[j, idx] : adj[i]) {
70             if (bel[i] < bel[j]) {
71                 g.edges.emplace_back(bel[i], bel[j]);
72             } else if (i < j) {
73                 g.cnte[bel[i]]++;
74             }
75         }
76     }
77     return g;
78 }
79 };

```

### 1.15.2 SCC

```

1 struct SCC {
2     i32 n;
3     std::vector<std::vector<i32>> adj;
4     std::vector<i32> dfn, low, bel;
5     std::vector<i32> stk;
6     i32 cur, cnt;

```

```

7     SCC() {}
8     SCC(i32 n) {
9         init(n);
10    }
11    void init(i32 n) {
12        this->n = n;
13        adj.assign(n, {});
14        dfn.assign(n, -1);
15        bel.assign(n, -1);
16        low.resize(n);
17        stk.clear();
18        cur = cnt = 0;
19    }
20    void add_edge(i32 u, i32 v) {
21        adj[u].push_back(v);
22    }
23    void dfs(i32 x) {
24        dfn[x] = low[x] = cur++;
25        stk.push_back(x);
26        for (auto y : adj[x]) {
27            if (dfn[y] == -1) {
28                dfs(y);
29                low[x] = std::min(low[x], low[y]);
30            } else if (bel[y] == -1) {
31                low[x] = std::min(low[x], dfn[y]);
32            }
33        }
34        if (dfn[x] == low[x]) {
35            i32 y;
36            do {
37                y = stk.back();
38                bel[y] = cnt;
39                stk.pop_back();
40            } while (y != x);
41            cnt++;
42        }
43    }
44    std::vector<i32> work() {
45        for (i32 i = 0; i < n; i++) {
46            if (dfn[i] == -1) {
47                dfs(i);
48            }
49        }
50        return bel;
51    }
52 };

```

## 1.16 网络流

### 1.16.1 Dinic

```
1 const i64 INF = 1E18;
2
3 struct Edge {
4     i32 u, v;
5     i64 cap, flow;
6 };
7
8 struct Dinic {
9     i32 n, s, t;
10    std::vector<Edge> edg;
11    std::vector<i32> dep, cur;
12    std::vector<std::vector<i32>> pos;
13    Dinic(i32 n) : n(n) {
14        dep.resize(n);
15        cur.resize(n);
16        pos.resize(n);
17    }
18    void addEdge(i32 u, i32 v, i64 w) {
19        edg.push_back({u, v, w, 0});
20        edg.push_back({v, u, 0, 0});
21        i32 m = edg.size();
22        pos[u].push_back(m - 2);
23        pos[v].push_back(m - 1);
24    }
25    i64 bfs() {
26        std::vector<bool> vis(n);
27        std::queue<i32> q;
28        q.push(s);
29        dep[s] = 0;
30        vis[s] = true;
31        while (!q.empty()) {
32            i32 now = q.front();
33            q.pop();
34            for (int i = 0; i < pos[now].size(); i++) {
35                auto &[u, v, cap, flow] = edg[pos[now][i]];
36                if (!vis[v] && cap > flow) {
37                    dep[v] = dep[u] + 1;
38                    vis[v] = true;
39                    q.push(v);
40                }
41            }
42        }
43        return vis[t];
44    }
45    i64 dfs(i32 now, i64 ctn) {
```

```
46    if (now == t || ctn == 0) {
47        return ctn;
48    }
49    i64 res = 0;
50    for (i32 i = cur[now]; i < pos[now].size(); ++i) {
51        auto &[u, v, cap, flow] = edg[pos[now][i]];
52        auto &[ru, rv, rcap, rflow] = edg[pos[now][i] ^ 1];
53        cur[now] = i;
54        if (dep[v] == dep[u] + 1 && cap > flow) {
55            i64 aug = dfs(v, std::min(ctn - res, cap - flow));
56            if (aug > 0) {
57                res += aug;
58                flow += aug;
59                rflow -= aug;
60                if (res == ctn) {
61                    return res;
62                }
63            }
64        }
65    }
66    return res;
67 }
68 i64 maxFlow(i32 s, i32 t) {
69     this->s = s;
70     this->t = t;
71     i64 res = 0;
72     while (bfs()) {
73         cur.assign(n, 0);
74         res += dfs(s, INF);
75     }
76     return res;
77 }
78 };
```

### 1.16.2 ISAP

```
1 const i64 INF = 1E18;
2
3 struct Edge {
4     i32 u, v;
5     i64 cap, flow;
6 };
7
8 struct ISAP {
9     i32 n, s, t;
10    std::vector<Edge> edg;
11    std::vector<i32> dep, rpos;
12    std::vector<std::vector<i32>> pos;
```



```

13 ISAP(i32 n) : n(n) {
14     rpos.resize(n);
15     pos.resize(n);
16     dep.resize(n);
17 }
18 void addEdge(i32 u, i32 v, i64 w) {
19     edg.push_back({u, v, w, 0});
20     edg.push_back({v, u, 0, 0});
21     i32 m = edg.size();
22     pos[u].push_back(m - 2);
23     pos[v].push_back(m - 1);
24 }
25 bool bfs() {
26     std::vector<bool> vis(n);
27     std::queue<i32> q;
28     q.push(t);
29     dep[t] = 0;
30     vis[t] = true;
31     while (!q.empty()) {
32         i32 now = q.front();
33         q.pop();
34         for (int i = 0; i < pos[now].size(); i++) {
35             auto &[u, v, cap, flow] = edg[pos[now][i] ^ 1];
36             if (!vis[u] && cap > flow) {
37                 dep[u] = dep[v] + 1;
38                 vis[u] = true;
39                 q.push(u);
40             }
41         }
42     }
43     return vis[s];
44 }
45 i64 augment() {
46     i64 res = INF;
47     i32 now = t;
48     while (now != s) {
49         auto &[u, v, cap, flow] = edg[rpos[now]];
50         res = std::min(res, cap - flow);
51         now = u;
52     }
53     now = t;
54     while (now != s) {
55         edg[rpos[now]].flow += res;
56         edg[rpos[now] ^ 1].flow -= res;
57         now = edg[rpos[now]].u;
58     }
59     return res;
60 }

```

```

61 i64 maxFlow(i32 s, i32 t) {
62     reset();
63     this->s = s;
64     this->t = t;
65     bfs();
66     std::vector<i32> num(n);
67     for (i32 i = 0; i < n; ++i) {
68         num[dep[i]]++;
69     }
70     i32 now = s;
71     i64 res = 0;
72     std::vector<i32> cur(n);
73     while (dep[s] < n) {
74         if (now == t) {
75             res += augment();
76             now = s;
77         }
78         bool flag = false;
79         for (i32 i = cur[now]; i < pos[now].size(); ++i) {
80             auto &[u, v, cap, flow] = edg[pos[now][i]];
81             if (cap > flow && dep[u] == dep[v] + 1) {
82                 flag = true;
83                 rpos[v] = pos[u][i];
84                 cur[u] = i;
85                 now = v;
86                 break;
87             }
88         }
89         if (!flag) {
90             i32 mnd = n - 1;
91             for (i32 i = 0; i < pos[now].size(); ++i) {
92                 auto &[u, v, cap, flow] = edg[pos[now][i]];
93                 if (cap > flow) {
94                     mnd = std::min(mnd, dep[v]);
95                 }
96             }
97             if (--num[dep[now]] == 0) {
98                 break;
99             }
100             dep[now] = mnd + 1;
101             num[dep[now]]++;
102             cur[now] = 0;
103             if (now != s) {
104                 now = edg[rpos[now]].u;
105             }
106         }
107     }
108     return res;

```

```

109     }
110     void reset() {
111         for (auto &e : edg) {
112             e.flow = 0;
113         }
114     }
115 };

```

### 1.16.3 Primal-Dual

```

1  const i64 INF = 1E18;
2
3  struct Edge {
4      i32 u, v;
5      i64 cap, flow, cost;
6  };
7
8  struct PrimalDual {
9      i32 n, s, t;
10     std::vector<Edge> edg;
11     std::vector<i64> pot, dis;
12     std::vector<std::pair<i32, i32>> dir;
13     std::vector<std::vector<i32>> pos;
14     PrimalDual(i32 n) : n(n) {
15         dis.resize(n);
16         dir.resize(n);
17         pot.resize(n);
18         pos.resize(n);
19     }
20     void addEdge(i32 u, i32 v, i64 w, i64 c) {
21         edg.push_back({u, v, w, 0, c});
22         edg.push_back({v, u, 0, 0, -c});
23         i32 m = edg.size();
24         pos[u].push_back(m - 2);
25         pos[v].push_back(m - 1);
26     }
27     void spfa() {
28         std::vector<bool> vis(n);
29         std::queue<i32> q;
30         q.push(s);
31         pot[s] = 0;
32         vis[s] = true;
33         while (!q.empty()) {
34             i32 now = q.front();
35             q.pop();
36             vis[now] = false;
37             for (i32 i = 0; i < pos[now].size(); ++i) {
38                 auto &[u, v, cap, flow, cost] = edg[pos[now][i]];

```

```

39             if (cap - flow && pot[v] > pot[u] + cost) {
40                 pot[v] = pot[u] + cost;
41                 if (!vis[v]) {
42                     vis[v] = true;
43                     q.push(v);
44                 }
45             }
46         }
47     }
48 }
49 bool dijkstra() {
50     auto cmp = [&](const std::pair<i64, i32> &a,
51                  const std::pair<i64, i32> &b) -> bool {
52         return a.first > b.first;
53     };
54     std::priority_queue<std::pair<i64, i32>,
55                       std::vector<std::pair<i64, i32>>,
56                       decltype(cmp)> pq(cmp);
57     std::vector<bool> vis(n);
58     dis.assign(n, INF);
59     dis[s] = 0;
60     pq.push({0, s});
61     while (!pq.empty()) {
62         i32 now = pq.top().second;
63         pq.pop();
64         if (vis[now]) {
65             continue;
66         }
67         vis[now] = true;
68         for (i32 i = 0; i < pos[now].size(); ++i) {
69             auto &[u, v, cap, flow, cost] = edg[pos[now][i]];
70             i64 pcost = cost + pot[u] - pot[v];
71             if (cap - flow > 0 && dis[v] > dis[u] + pcost) {
72                 dir[v] = {u, pos[now][i]};
73                 dis[v] = dis[u] + pcost;
74                 if (!vis[v]) {
75                     pq.push({dis[v], v});
76                 }
77             }
78         }
79     }
80     return dis[t] < INF;
81 }
82 std::pair<i64, i64> work(i32 s, i32 t) {
83     reset();
84     this->s = s;
85     this->t = t;
86     spfa();

```

```

87     i64 mnc = 0;
88     i64 mxf = 0;
89     while (dijkstra()) {
90         i64 aug = INF;
91         for (i32 i = 0; i < n; ++i) {
92             pot[i] += dis[i];
93         }
94         for (i32 i = t; i != s; i = dir[i].first) {
95             auto &[u, v, cap, flow, cost] = edg[dir[i].second];
96             aug = std::min(aug, cap - flow);
97         }
98         for (i32 i = t; i != s; i = dir[i].first) {
99             edg[dir[i].second ^ 1].flow -= aug;
100             edg[dir[i].second].flow += aug;
101         }
102         mnc += aug * pot[t];
103         mxf += aug;
104     }
105     return std::make_pair(mxf, mnc);
106 }
107 void reset() {
108     for (auto &e : edg) {
109         e.flow = 0;
110     }
111     pot.assign(n, INF);
112 }
113 };

```

#### 1.16.4 Stoer-Wagner

```

1  const i64 INF = 1E18;
2
3  struct StoerWagner {
4      i32 n;
5      std::vector<std::vector<i64>> adj;
6      StoerWagner(i32 n) : n(n) {
7          adj = std::vector(n, std::vector<i64>(n, 0));
8      }
9      void addEdge(i32 u, i32 v, i64 w) {
10         adj[u][v] += w;
11         adj[v][u] += w;
12     }
13     i64 work() {
14         i64 res = INF;
15         std::vector<bool> in(n);
16         std::vector<i32> bel(n);
17         std::iota(bel.begin(), bel.end(), 0);
18         for (i32 i = 0; i < n - 1; ++i) {

```

```

19         std::vector<bool> vis(n);
20         std::vector<i64> wei(n);
21         i32 lst = -1;
22         for (i32 j = 0; j < n - i - 1; ++j) {
23             i32 cur = -1;
24             for (i32 k = 0; k < n; ++k) {
25                 if (!in[k] && !vis[k] && (cur == -1 || wei[k] > wei[cur]))
26                     cur = k;
27             }
28             vis[cur] = true;
29             for (i32 k = 0; k < n; ++k) {
30                 if (!in[k] && !vis[k]) {
31                     wei[k] += adj[cur][k];
32                 }
33             }
34             lst = cur;
35         }
36         i32 cur = -1;
37         for (i32 k = 0; k < n; ++k) {
38             if (!in[k] && !vis[k]) {
39                 cur = k;
40                 break;
41             }
42         }
43         res = std::min(res, wei[cur]);
44         in[cur] = true;
45         for (i32 k = 0; k < n; ++k) {
46             if (!in[k]) {
47                 adj[lst][k] += adj[cur][k];
48                 adj[k][lst] += adj[k][cur];
49             }
50         }
51         bel[cur] = lst;
52     }
53     return res;
54 }
55 };
56 };

```

## 2 字符串

### 2.1 Duo-Hashing

```

1  const i64 MOD1 = 212370440130137957;
2  const i64 MOD2 = 1e9 + 7;
3  const i64 BASE1 = 127;

```

```

4 const i64 BASE2 = 131;
5
6 i64 power(i64 a, i64 b, i64 p) {
7     i64 res = 1;
8     while (b) {
9         if (b & 1) res = res * a % p;
10        a = a * a % p;
11        b >>= 1;
12    }
13    return res;
14 }
15
16 struct Hashing {
17     std::vector<i64> h1, h2;
18     std::string s;
19     Hashing() {}
20     Hashing(const std::string &s) : s(s) {
21         i32 n = s.length();
22         h1.resize(n + 1);
23         h2.resize(n + 1);
24         for (i32 i = 0; i < n; ++i) {
25             h1[i + 1] = (h1[i] * BASE1 + (i64)s[i]) % MOD1;
26             h2[i + 1] = (h2[i] * BASE2 + (i64)s[i]) % MOD2;
27         }
28     }
29     auto getHash(i32 l, i32 r) {
30         i64 res1 = (h1[r + 1] - h1[l] * power(BASE1, r - l + 1, MOD1) % MOD1 +
31             MOD1) % MOD1;
32         i64 res2 = (h2[r + 1] - h2[l] * power(BASE2, r - l + 1, MOD2) % MOD2 +
33             MOD2) % MOD2;
34         return std::make_pair(res1, res2);
35     }
36     bool hcmp(Hashing &u, i32 l, i32 r) {
37         return getHash(l, r) == u.getHash(l, r);
38     }
39 };

```

## 2.2 KMP

```

1 std::vector<i32> getPre(std::string s) {
2     i32 n = s.length();
3     std::vector<i32> res(n);
4     for (i32 i = 1; i < n; ++i) {
5         i32 j = res[i - 1];
6         while (j > 0 && s[i] != s[j]) {
7             j = res[j - 1];
8         }
9         if (s[i] == s[j]) {

```

```

10            j++;
11        }
12        res[i] = j;
13    }
14    return res;
15 }
16
17 std::vector<i32> KMP(std::string p, std::string s) {
18     i32 n = p.length();
19     i32 m = s.length();
20     std::string cur = p + '#' + s;
21     std::vector<i32> lps = getPre(cur);
22     std::vector<i32> res;
23     for (i32 i = n + 1; i <= n + m; ++i) {
24         if (lps[i] == n) {
25             res.push_back(i - 2 * n);
26         }
27     }
28     return res;
29 }

```

## 2.3 Manacher

```

1 std::vector<i32> manacher(std::string s) {
2     std::string t = "#";
3     for (auto c : s) {
4         t += c;
5         t += '#';
6     }
7     i32 n = t.size();
8     std::vector<i32> r(n);
9     for (i32 i = 0, j = 0; i < n; ++i) {
10        if (2 * j - i >= 0 && j + r[j] > i) {
11            r[i] = std::min(r[2 * j - i], j + r[j] - i);
12        }
13        while (i - r[i] >= 0 && i + r[i] < n && t[i - r[i]] == t[i + r[i]]) {
14            r[i] += 1;
15        }
16        if (i + r[i] > j + r[j]) {
17            j = i;
18        }
19    }
20    return r;
21 }

```

## 2.4 Z-Function

```

1 std::vector<i32> Z(std::string s) {
2     i32 n = s.size();
3     std::vector<i32> z(n + 1);
4     z[0] = n;
5     for (i32 i = 1, j = 1; i < n; i++) {
6         z[i] = std::max(0, std::min(j + z[j] - i, z[i - j]));
7         while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
8             z[i]++;
9         }
10        if (i + z[i] > j + z[j]) {
11            j = i;
12        }
13    }
14    return z;
15 }

```

## 3 数学

### 3.1 CatalanNumbers

```

1 const i64 MOD = 1E9 + 7;
2
3 std::vector<i32> cat;
4
5 void catalan(i32 n) {
6     cat.resize(n + 1);
7     cat[0] = cat[1] = 1;
8     for (i32 i = 2; i <= n; ++i) {
9         cat[i] = cat[i - 1] * ((4 * i % MOD - 2 + MOD) % MOD) % MOD * inv(i +
10         1, MOD) % MOD;
11     }
12
13 // catalan-number:
14 //     H[i] = H[i - 1] * (4n - 2) / (n + 1)
15 //     H[i] = C(2n, n) / (n + 1)

```

### 3.2 Combination-Dynamic

```

1 const i64 MOD = 1E9 + 7;
2
3 i64 power(i64 a, i64 b, i64 p = MOD) {
4     i64 res = 1;
5     while (b) {
6         if (b & 1) res = res * a % p;
7         a = a * a % p;

```

```

8         b >>= 1;
9     }
10    return res;
11 }
12
13 i64 inv(i64 a, i64 p = MOD) {
14     return power(a, p - 2, p);
15 }
16
17 template<typename T>
18 struct Comb {
19     i32 n;
20     i64 mod = 1E9 + 7;
21     std::vector<T> _fac, _invfac, _inv;
22     Comb() : n(0) {
23         _fac = _invfac = {1};
24         _inv = {0};
25     }
26     Comb(i32 n, i64 mod) : Comb() {
27         this->mod = mod;
28         init(n);
29     }
30     void init(i32 m) {
31         if (n >= m) return;
32         _invfac.resize(m + 1);
33         _fac.resize(m + 1);
34         _inv.resize(m + 1);
35         for (i32 i = n + 1; i <= m; ++i) {
36             _fac[i] = _fac[i - 1] * i % mod;
37         }
38         _invfac[m] = inv(_fac[m], mod);
39         for (i32 i = m; i > n; --i) {
40             _invfac[i - 1] = _invfac[i] * i % mod;
41             _inv[i] = _fac[i - 1] * _invfac[i] % mod;
42         }
43         n = m;
44     }
45     T operator () (i32 a, i32 b) {
46         if (a < b || b < 0) return T{};
47         else return fac(a) * invfac(a - b) % mod * invfac(b) % mod;
48     }
49     T invfac(i32 a) {
50         if (a > n) init(2 * a);
51         return _invfac[a];
52     }
53     T fac(i32 a) {
54         if (a > n) init(2 * a);
55         return _fac[a];

```

```

56     }
57     T civ(i32 a) {
58         if (a > n) init(2 * a);
59         return _inv[a];
60     }
61 };

```

### 3.3 Combination

```

1  const i64 MOD = 1E9 + 7;
2
3  template<typename T>
4  struct Comb {
5      T n, mod;
6      std::vector<T> fac, inv, invfac;
7      Comb(T n = 0, T mod = 1E9 + 7) {
8          init(n, mod);
9      }
10     void init(i32 n, i32 mod) {
11         this->n = n;
12         this->mod = mod;
13         invfac.assign(n, T{});
14         inv.assign(n, T{});
15         fac.assign(n, T{});
16         fac[0] = fac[1] = 1;
17         inv[1] = 1;
18         invfac[0] = invfac[1] = 1;
19         for (i64 i = 2; i < n; ++i) {
20             inv[i] = (mod - mod / i) * inv[mod % i] % mod;
21             fac[i] = fac[i - 1] * i % mod;
22             invfac[i] = invfac[i - 1] * inv[i] % mod;
23         }
24     }
25     T comb(i32 a, i32 b) {
26         if(a < b) return T{};
27         else return fac[a] * invfac[a - b] % mod * invfac[b] % mod;
28     }
29 };

```

/\*

definition:

$A(n, m) = n! / (n - m)!$   
 $C(n, m) = n! / ((n - m)! * m!)$

recursion:

$C(n, m) = C(n, m - 1) * (n - m + 1) / m$   
 $C(n, m) = C(n - 1, m) + C(n - 1, m - 1)$

```

40
41 property:
42     C(n + m + 1, n) = sigma[i = 0 -> m] { C(n + i, i) }
43     C(n, m) * C(m, r) = C(n, r) * C(n - r, m - r)
44
45     sigma[i = 0 -> n] { C(n, i) } = 2 ^ n
46     sigma[i = 0 -> n] { C(n, i) * (x ^ i) } = (x + 1) ^ n
47
48     sigma[i = 0 -> n] { (-1) ^ i * C(n, i) } = 0;
49     C(n, 0) + C(n, 2) + ... = C(n, 1) + C(n, 3) + ... = 2 ^ (n - 1)
50
51     C(n + m, r) = sigma[i = 0 -> min{n, m, r}] { C(n, i) * C(m, r - i) }
52
53     m * C(n, m) = n * C(n - 1, m - 1)
54
55     sigma[i = 0 -> n] { C(n, i) * i } = n * 2 ^ (n - 1)
56     sigma[i = 0 -> n] { C(n, i) * i ^ 2 } = n * (n + 1) * 2 ^ (n - 2)
57
58     sigma[i = 0 -> n] { C(n, i) ^ 2 } = C(2 * n, n)
59
60 */

```

### 3.4 Matrix

```

1  const i64 MOD = 998244353;
2
3  i64 power(i64 a, i64 b, i64 p = MOD) {
4      i64 res = 1;
5      while (b) {
6          if (b & 1) res = res * a % p;
7          a = a * a % p;
8          b >>= 1;
9      }
10     return res;
11 }
12
13 i64 inv(i64 a, i64 p = MOD) {
14     return power(a, p - 2, p);
15 }
16
17 template <typename T>
18 struct Matrix {
19     i32 n, m;
20     std::vector<std::vector<T>>> v;
21     Matrix(i32 n, i32 m) : n(n), m(m) {
22         v = std::vector(n, std::vector<T>(m));
23     }
24     Matrix(const std::vector<std::vector<T>>> &v) : Matrix(v.size(), v[0].size())

```

```

25     ) {
26         for (i32 i = 0; i < n; ++i) {
27             for (i32 j = 0; j < m; ++j) {
28                 this->v[i][j] = v[i][j];
29             }
30         }
31     std::vector<T>& operator [] (i32 x) {
32         assert(x < n);
33         return v[x];
34     }
35     Matrix<T> operator = (const Matrix<T> &x) {
36         n = x.n;
37         m = x.m;
38         v = x;
39         return *this;
40     }
41     Matrix<T> operator + (const Matrix<T> &x) {
42         assert(n == x.n && m == x.m);
43         Matrix<T> res(n, m);
44         for (i32 i = 0; i < n; ++i) {
45             for (i32 j = 0; j < m; ++j) {
46                 res[i][j] = v[i][j] + v[i][j];
47             }
48         }
49         return res;
50     }
51     Matrix<T> operator += (const Matrix<T> &x) {
52         return *this = *this + x;
53     }
54     Matrix<T> operator - (const Matrix<T> &x) {
55         assert(n == x.n && m == x.m);
56         Matrix<T> res(n, m);
57         for (i32 i = 0; i < n; ++i) {
58             for (i32 j = 0; j < m; ++j) {
59                 res[i][j] = v[i][j] - v[i][j];
60             }
61         }
62         return res;
63     }
64     Matrix<T> operator -= (const Matrix<T> &x) {
65         return *this = *this - x;
66     }
67     Matrix<T> operator * (const Matrix<T> &x) {
68         assert(m == x.n);
69         Matrix<T> res(n, x.m);
70         for (i32 i = 0; i < n; ++i) {
71             for (i32 j = 0; j < x.m; ++j) {

```

```

72                 for (i32 k = 0; k < m; ++k) {
73                     res[i][j] = res[i][j] + v[i][k] * v[k][j];
74                 }
75             }
76         }
77         return res;
78     }
79     Matrix<T> operator *= (const Matrix<T> &x) {
80         return *this = *this * x;
81     }
82     static Matrix<T> power(Matrix<T> a, i64 b) {
83         Matrix<T> res(Matrix::eye(a.n));
84         while (b > 0) {
85             if (b & 1) res = res * a;
86             a = a * a;
87             b >>= 1;
88         }
89         return res;
90     }
91     static Matrix<T> eye(i32 n) {
92         Matrix<T> res(n, n);
93         for (i32 i = 0; i < n; ++i) {
94             res[i][i] = 1;
95         }
96         return res;
97     }
98     friend std::ostream& operator << (std::ostream &os, const Matrix<T> &x) {
99         for (i32 i = 0; i < x.n; ++i) {
100             for (i32 j = 0; j < x.m; ++j) {
101                 os << x.v[i][j] << " \n"[j + 1 == x.m];
102             }
103         }
104         return os;
105     }
106 };
107
108 template <typename T>
109 i64 det(Matrix<T> x) {
110     assert(x.n == x.m);
111     i64 res = 1;
112     for (i32 i = 0; i < x.n; ++i) {
113         i32 pivot = i;
114         for (i32 j = i; j < x.n; ++j) {
115             if (x[j][i] != 0) {
116                 pivot = j;
117                 break;
118             }
119         }

```

```

120     if (x[pivot][i] == 0) {
121         return 0;
122     }
123     if (pivot != i) {
124         std::swap(x[i], x[pivot]);
125         res = (MOD - res) % MOD;
126     }
127     for (i32 j = i + 1; j < x.n; ++j) {
128         if (x[j][i] == 0) continue;
129         while (x[i][j] != 0) {
130             i64 fix = x[j][i] / x[i][i];
131             for (i32 k = i; k < x.n; ++k) {
132                 x[j][k] = ((x[j][k] - fix * x[i][k]) % MOD + MOD) % MOD;
133             }
134             for (i32 k = i; k < x.n; ++k) {
135                 std::swap(x[i][k], x[j][k]);
136             }
137             res = (MOD - res) % MOD;
138         }
139         for (i32 k = i; k < x.n; ++k) {
140             std::swap(x[i][k], x[j][k]);
141         }
142         res = (MOD - res) % MOD;
143     }
144     res = res * x[i][i] % MOD;
145 }
146 return res;
147 }

```

### 3.5 Power-Inv

```

1 const i64 MOD = 1E9 + 7;
2
3 i64 power(i64 a, i64 b, i64 p = MOD) {
4     i64 res = 1;
5     while (b) {
6         if (b & 1) res = res * a % p;
7         a = a * a % p;
8         b >>= 1;
9     }
10    return res;
11 }
12
13 i64 inv(i64 a, i64 p = MOD) {
14     return power(a, p - 2, p);
15 }
16
17 i64 pow(i64 a, i64 b, i64 p = MOD) {

```

```

18     if (b < 0) {
19         return power(inv(a, p), std::abs(b), p);
20     } else {
21         return power(a, b, p);
22     }
23 }

```

### 3.6 Xor

```

1 i64 rangeXor(i64 a, i64 b) {
2     std::function<i64(i64)> pref = [](i64 x) {
3         if (x % 4 == 0) {
4             return x;
5         } else if (x % 4 == 1) {
6             return 1LL;
7         } else if (x % 4 == 2) {
8             return x + 1;
9         } else {
10            return 0LL;
11        }
12    };
13    return pref(b) ^ pref(a - 1);
14 };

```

## 4 数据结构

### 4.1 01-Trie

```

1 const i32 MAXH = 30;
2
3 struct Trie {
4     i32 n, tot, root;
5     std::vector<i32> w, sub;
6     std::vector<std::array<i32, 2>> chd;
7     Trie(i32 n) : n(n), tot(0), root(0) {
8         i32 siz = n * (i32)std::ceil(std::lg(n));
9         w.assign(siz, 0);
10        sub.assign(siz, 0);
11        chd.assign(siz, {0, 0});
12    }
13    void maintain(i32 x) {
14        w[x] = sub[x] = 0;
15        if (chd[x][0]) {
16            w[x] += w[chd[x][0]];
17            sub[x] ^= sub[chd[x][0]] << 1;
18        }

```



```

19     if (chd[x][1]) {
20         w[x] += w[chd[x][1]];
21         sub[x] ^= (sub[chd[x][1]] << 1) | (w[chd[x][1]] & 1);
22     }
23     w[x] = w[x] & 1;
24 }
25 void insert(i32 &p, i32 x, i32 dep) {
26     if (!p) {
27         p = ++tot;
28     }
29     if (dep >= MAXH) {
30         w[p]++;
31         return;
32     }
33     insert(chd[p][x & 1], x >> 1, dep + 1);
34     maintain(p);
35 }
36 void insert(i32 x) {
37     insert(root, x, 0);
38 }
39 void erase(i32 &p, i32 x, i32 dep) {
40     if (dep >= MAXH) {
41         w[p]--;
42         return;
43     }
44     erase(chd[p][x & 1], x >> 1, dep + 1);
45     maintain(p);
46 }
47 void erase(i32 x) {
48     erase(root, x, 0);
49 }
50 void addAll(i32 x = 0) {
51     std::swap(chd[x][0], chd[x][1]);
52     if (chd[x][0]) addAll(chd[x][0]);
53     maintain(x);
54 }
55 i32 merge(Trie u, i32 a = 0, i32 b = 0) {
56     if (!a) return b;
57     if (!b) return a;
58     w[a] = w[a] + u.w[b];
59     sub[a] = sub[a] ^ u.sub[b];
60     chd[a][0] = merge(u, chd[a][0], u.chd[b][0]);
61     chd[a][1] = merge(u, chd[a][1], u.chd[b][1]);
62     return a;
63 }
64 };

```

## 4.2 LeftistTree

```

1  template <typename Info, typename Tag, typename Func = std::less<Info>>
2  struct LeftistTree {
3      struct Node {
4          Info info;
5          Tag tag;
6          i32 dis = 0;
7          Node *lc = nullptr;
8          Node *rc = nullptr;
9          Node(const Info &x) : info(x) {}
10     } *root = nullptr;
11     i32 siz = 0;
12     Func cmp;
13     LeftistTree() = default;
14     LeftistTree(Func cmp) : cmp(cmp) {}
15     ~LeftistTree() {
16         clear();
17     }
18     void abdicate() {
19         root = nullptr;
20         siz = 0;
21     }
22     i32 dist(Node *x) {
23         if (x == nullptr) {
24             return -1;
25         }
26         return x->dis;
27     }
28     void pushdown(Node *x) {
29         if (x == nullptr) {
30             return;
31         }
32         x->info.apply(x->tag);
33         if (x->lc != nullptr) {
34             x->lc->tag.apply(x->tag);
35         }
36         if (x->rc != nullptr) {
37             x->rc->tag.apply(x->tag);
38         }
39         x->tag = Tag{};
40     }
41     Node* merge(Node* x, Node* y) {
42         if (x == nullptr) return y;
43         if (y == nullptr) return x;
44         pushdown(x);
45         pushdown(y);
46         if (cmp(x->info, y->info)) {

```

```

47     std::swap(x, y);
48 }
49 x->rc = merge(x->rc, y);
50 if (dist(x->lc) < dist(x->rc)) {
51     std::swap(x->lc, x->rc);
52 }
53 x->dis = dist(x->rc) + 1;
54 return x;
55 }
56 void merge(LeftistTree &x) {
57     root = merge(root, x.root);
58     siz += x.size();
59     x.abdicate();
60 }
61 void clear() {
62     if (root == nullptr) return;
63     std::queue<Node*> q;
64     q.push(root);
65     while (!q.empty()) {
66         Node *u = q.front();
67         q.pop();
68         if (u->lc != nullptr) {
69             q.push(u->lc);
70         }
71         if (u->rc != nullptr) {
72             q.push(u->rc);
73         }
74         delete u;
75     }
76     abdicate();
77 }
78 void apply(const Tag &v) {
79     if (root != nullptr) {
80         root->tag.apply(v);
81     }
82 }
83 void push(Info x) {
84     Node *temp = new Node(x);
85     root = merge(root, temp);
86     siz++;
87 }
88 void pop() {
89     pushdown(root);
90     Node *temp = root;
91     root = merge(root->lc, root->rc);
92     delete temp;
93     siz--;
94 }

```

```

95 bool empty() {
96     return root == nullptr;
97 }
98 Info top() {
99     pushdown(root);
100     return root->info;
101 }
102 i32 size() {
103     return siz;
104 }
105 };
106
107 struct Tag {
108     i64 add = 0;
109     void apply(const Tag &v) {
110         add += v.add;
111     }
112 };
113
114 struct Info {
115     i64 val = 0;
116     void apply(const Tag &v) {
117         val += v.add;
118     }
119     bool operator < (const Info &u) const {
120         return val < u.val;
121     }
122     bool operator > (const Info &u) const {
123         return val > u.val;
124     }
125 };

```

### 4.3 SparseTableByEnar

```

1 template <typename T, typename Func = std::function<T(const T&, const T&)>>
2 struct ST {
3     ST(const std::vector<T> &v, Func func =
4         [](const T& a, const T& b) {
5             return std::max(a, b);
6         }) : func(std::move(func)) {
7         int k = std::__lg(v.size());
8         st = std::vector<std::vector<T>>(k + 1, std::vector<T>(v.size()));
9         st[0] = v;
10        for(int i = 0; i < k; ++i) {
11            for(int j = 0; j + (1 << (i + 1)) - 1 < v.size(); ++j) {
12                st[i + 1][j] = this->func(st[i][j], st[i][j + (1 << i)]);
13            }
14        }

```

```

15     }
16 }
17 T range(int l, int r) {
18     int t = std::__lg(r - l + 1);
19     return func(st[t][l], st[t][r + 1 - (1 << t)]);
20 }
21 std::vector<std::vector<T>> st;
22 Func func;
23 };

```

## 4.4 并查集

### 4.4.1 DSU-Rollback

```

1 struct DSU {
2     std::vector<i32> p, siz;
3     std::vector<std::array<i32, 2>> his;
4     DSU(i32 n) : siz(n + 1, 1), p(n + 1) {
5         std::iota(p.begin(), p.end(), 0);
6     }
7     i32 find(i32 x) {
8         while (p[x] != x) {
9             x = p[x];
10        }
11        return x;
12    }
13    bool merge(i32 x, i32 y) {
14        x = find(x);
15        y = find(y);
16        if (x == y) {
17            return false;
18        }
19        if (siz[x] < siz[y]) {
20            std::swap(x, y);
21        }
22        his.push_back({x, y});
23        siz[x] += siz[y];
24        p[y] = x;
25        return true;
26    }
27    i32 time() {
28        return his.size();
29    }
30    void revert(i32 tm) {
31        while (his.size() > tm) {
32            auto [x, y] = his.back();
33            his.pop_back();
34            p[y] = y;

```

```

35        siz[x] -= siz[y];
36    }
37 }
38 };

```

### 4.4.2 DSU

```

1 struct DSU {
2     std::vector<i32> p, siz;
3     DSU() {}
4     DSU(i32 n) {
5         init(n);
6     }
7     void init(i32 n) {
8         p.resize(n);
9         siz.assign(n, 1);
10        std::iota(p.begin(), p.end(), 0);
11    }
12    i32 find(i32 x) {
13        while (x != p[x]) x = p[x] = p[p[x]];
14        return x;
15    }
16    bool same(i32 x, i32 y) {
17        return find(x) == find(y);
18    }
19    bool merge(i32 x, i32 y) {
20        x = find(x);
21        y = find(y);
22        if (x == y) {
23            return false;
24        } else {
25            p[y] = x;
26            siz[x] += siz[y];
27            return true;
28        }
29    }
30    i32 size(i32 x) {
31        return siz[find(x)];
32    }
33 };

```

## 4.5 树状数组

### 4.5.1 Fenwick-Range

```

1 template<typename T>
2 struct Fenwick {
3     i32 n;

```

```

4  std::vector<T> s, t;
5  Fenwick(i32 n) : n(n) {
6      s.assign(n, T{});
7      t.assign(n, T{});
8  }
9  void baseApply(i32 x, const T &v) {
10     for (i32 i = x + 1; i <= n; i += i & -i) {
11         s[i - 1] = s[i - 1] + v;
12         t[i - 1] = t[i - 1] + x * v;
13     }
14 }
15 void rangeApply(i32 l, i32 r, const T &v) {
16     baseApply(l, v);
17     baseApply(r + 1, -v);
18 }
19 void apply(i32 x, const T &v) {
20     rangeApply(x, x, v);
21 }
22 T baseQuery(i32 x) {
23     T res{};
24     for (i32 i = x; i > 0; i -= i & -i) {
25         res = res + x * s[i - 1] - t[i - 1];
26     }
27     return res;
28 }
29 T rangeQuery(i32 l, i32 r) {
30     return baseQuery(r) - baseQuery(l - 1);
31 }
32 T query(i32 x) {
33     return rangeQuery(x, x);
34 }
35 };

```

## 4.5.2 Fenwick

```

1  template<typename T>
2  struct Fenwick {
3      i32 n;
4      std::vector<T> tree;
5      Fenwick(i32 n = 0) {
6          init(n);
7      }
8      void init(i32 n) {
9          this->n = n;
10         tree.assign(n, T{});
11     }
12     void apply(i32 x, const T &v) {
13         for (i32 i = x + 1; i <= n; i += i & -i) {

```

```

14         tree[i - 1] = tree[i - 1] + v;
15     }
16 }
17 T query(i32 x) {
18     T res{};
19     for (i32 i = x; i > 0; i -= i & -i) {
20         res = res + tree[i - 1];
21     }
22     return res;
23 }
24 T rangeQuery(i32 l, i32 r) {
25     return query(r) - query(l);
26 }
27 };

```

## 4.6 线段树

### 4.6.1 LazySegmentTree-Extra

```

1  const i64 INF = 1E18;
2  const i64 MOD = 1E9 + 7;
3
4  template <typename Info, typename Tag>
5  struct SegmentTree {
6      i32 n;
7      std::vector<Tag> tag;
8      std::vector<Info> info;
9      SegmentTree(i32 n) : n(n), info(4 << std::__lg(n)), tag(4 << std::__lg(n)) {}
10     SegmentTree(const std::vector<auto> &v) : SegmentTree(v.size()) {
11         auto build = [&](auto self, i32 p, i32 l, i32 r) {
12             if(r == l) {
13                 info[p].init(v[l]);
14                 return;
15             }
16             i32 m = (l + r) / 2;
17             self(self, p << 1, l, m);
18             self(self, p << 1 | 1, m + 1, r);
19             pull(p);
20         };
21         build(build, 1, 0, n - 1);
22     }
23     void add(i32 p, const Tag &v) {
24         info[p].apply(v);
25         tag[p].apply(v);
26     }
27     void mul(i32 p, const Tag &v) {
28         info[p].multi(v);

```

```

29     tag[p].multi(v);
30 }
31 void update(i32 p, const Tag &v) {
32     info[p].update(v);
33     tag[p].update(v);
34 }
35 void pull(i32 p) {
36     info[p] = info[p << 1] + info[p << 1 | 1];
37 }
38 void push(i32 p) {
39     update(p << 1, tag[p]);
40     update(p << 1 | 1, tag[p]);
41     tag[p] = Tag();
42 }
43 void rangeApply(i32 p, i32 x, i32 y, i32 l, i32 r, const Tag &v) {
44     if (x <= l && y >= r) {
45         add(p, v);
46         return;
47     }
48     push(p);
49     i32 m = (l + r) / 2;
50     if (x <= m) rangeApply(p << 1, x, y, l, m, v);
51     if (y > m) rangeApply(p << 1 | 1, x, y, m + 1, r, v);
52     pull(p);
53 }
54 void rangeApply(i32 x, i32 y, const Tag &v) {
55     rangeApply(1, x, y, 0, n - 1, v);
56 }
57 void apply(i32 x, const Tag &v) {
58     rangeApply(1, x, x, 0, n - 1, v);
59 }
60 void rangeMultiply(i32 p, i32 x, i32 y, i32 l, i32 r, const Tag &v) {
61     if (x <= l && y >= r) {
62         mul(p, v);
63         return;
64     }
65     push(p);
66     i32 m = (l + r) / 2;
67     if (x <= m) rangeMultiply(p << 1, x, y, l, m, v);
68     if (y > m) rangeMultiply(p << 1 | 1, x, y, m + 1, r, v);
69     pull(p);
70 }
71 void rangeMultiply(i32 x, i32 y, const Tag &v) {
72     rangeMultiply(1, x, y, 0, n - 1, v);
73 }
74 void multiply(i32 x, const Tag &v) {
75     rangeMultiply(1, x, x, 0, n - 1, v);
76 }

```

```

77 Info rangeQuery(i32 p, i32 x, i32 y, i32 l, i32 r) {
78     if (x <= l && y >= r) return info[p];
79     Info res;
80     push(p);
81     i32 m = (l + r) / 2;
82     if (x <= m) res = res + rangeQuery(p << 1, x, y, l, m);
83     if (y > m) res = res + rangeQuery(p << 1 | 1, x, y, m + 1, r);
84     pull(p);
85     return res;
86 }
87 Info rangeQuery(i32 x, i32 y) {
88     return rangeQuery(1, x, y, 0, n - 1);
89 }
90 Info query(i32 x) {
91     return rangeQuery(1, x, x, 0, n - 1);
92 }
93 };
94
95 struct Tag {
96     i64 add = 0;
97     i64 mul = 1;
98     void apply(const Tag &v) {
99         add = (add + v.add) % MOD;
100     }
101     void multi(const Tag &v) {
102         mul = mul * v.mul % MOD;
103         add = add * v.mul % MOD;
104     }
105     void update(const Tag &v) {
106         multi(v);
107         apply(v);
108     }
109 };
110
111 struct Info {
112     i64 sum = 0;
113     i64 len = 1;
114     void init(const i64 &x) {
115         sum = x;
116     }
117     void apply(const Tag &v) {
118         sum = (sum + len * v.add % MOD) % MOD;
119     }
120     void multi(const Tag &v) {
121         sum = sum * v.mul % MOD;
122     }
123     void update(const Tag &v) {
124         multi(v);

```

```

125     apply(v);
126 }
127 Info operator + (const Info &a) {
128     Info res;
129     res.sum = (sum + a.sum) % MOD;
130     res.len = (len + a.len) % MOD;
131     return res;
132 }
133 };

```

#### 4.6.2 LazySegmentTree-Short

```

1 template <typename T>
2 struct LazySegmentTree {
3     i32 n;
4     std::vector<T> tree, tag;
5     LazySegmentTree(i32 n) : n(n), tree(4 << std::__lg(n)), tag(4 << std::__lg(n)) {}
6     LazySegmentTree(std::vector<T> &v) : LazySegmentTree(v.size()) {
7         auto build = [&](auto &&build, i32 p, i32 l, i32 r) {
8             if(r == l) {
9                 tree[p] = v[l];
10                return;
11            }
12            i32 m = (l + r) / 2;
13            build(p << 1, l, m);
14            build(p << 1 | 1, m + 1, r);
15            pull(p);
16        };
17        build(1, 0, n - 1);
18    }
19    void pull(i32 p) {
20        tree[p] = tree[p << 1] + tree[p << 1 | 1];
21    }
22    void push(i32 p, i32 l, i32 r) {
23        if (l != r && tag[p] > 0) {
24            i32 m = (l + r) / 2;
25            tag[p << 1] += tag[p];
26            tag[p << 1 | 1] += tag[p];
27            tree[p << 1] += tag[p] * (m - l + 1);
28            tree[p << 1 | 1] += tag[p] * (r - m);
29            tag[p] = 0;
30        }
31    }
32    void rangeApply(i32 p, i32 x, i32 y, i32 l, i32 r, const T &v) {
33        if (x <= l && y >= r) {
34            tree[p] += (r - l + 1) * v;
35            tag[p] += v;

```

```

36        return;
37    }
38    push(p, l, r);
39    i32 m = (l + r) / 2;
40    if (x <= m) rangeApply(p << 1, x, y, l, m, v);
41    if (y > m) rangeApply(p << 1 | 1, x, y, m + 1, r, v);
42    pull(p);
43 }
44 void rangeApply(i32 x, i32 y, const T &v) {
45     rangeApply(1, x, y, 0, n - 1, v);
46 }
47 void apply(i32 x, const T &v) {
48     rangeApply(1, x, x, 0, n - 1, v);
49 }
50 T rangeQuery(i32 p, i32 x, i32 y, i32 l, i32 r) {
51     if (x <= l && y >= r) return tree[p];
52     T res = 0;
53     push(p, l, r);
54     i32 m = (l + r) / 2;
55     if (x <= m) res += rangeQuery(p << 1, x, y, l, m);
56     if (y > m) res += rangeQuery(p << 1 | 1, x, y, m + 1, r);
57     pull(p);
58     return res;
59 }
60 T rangeQuery(i32 x, i32 y) {
61     return rangeQuery(1, x, y, 0, n - 1);
62 }
63 T query(i32 x) {
64     return rangeQuery(1, x, x, 0, n - 1);
65 }
66 };

```

#### 4.6.3 LazySegmentTree

```

1 const i64 INF = 1E18;
2
3 template<typename Info, typename Tag>
4 struct SegmentTree {
5     i32 n;
6     std::vector<Tag> tag;
7     std::vector<Info> info;
8     SegmentTree(i32 n) : n(n), info(4 << std::__lg(n)), tag(4 << std::__lg(n)) {}
9     SegmentTree(const std::vector<auto> &v) : SegmentTree(v.size()) {
10         auto build = [&](auto self, i32 p, i32 l, i32 r) {
11             if(r == l) {
12                 info[p].init(v[l]);
13                 return;

```

```

14     }
15     i32 m = (l + r) / 2;
16     self(self, p << 1, l, m);
17     self(self, p << 1 | 1, m + 1, r);
18     pull(p);
19 };
20 build(build, 1, 0, n - 1);
21 }
22 void update(i32 p, const Tag &v) {
23     info[p].apply(v);
24     tag[p].apply(v);
25 }
26 void pull(i32 p) {
27     info[p] = info[p << 1] + info[p << 1 | 1];
28 }
29 void push(i32 p, i32 l, i32 r) {
30     update(p << 1, tag[p]);
31     update(p << 1 | 1, tag[p]);
32     tag[p] = Tag();
33 }
34 void rangeApply(i32 p, i32 x, i32 y, i32 l, i32 r, const Tag &v) {
35     if (x <= l && y >= r) {
36         update(p, v);
37         return;
38     }
39     push(p, l, r);
40     i32 m = (l + r) / 2;
41     if (x <= m) rangeApply(p << 1, x, y, l, m, v);
42     if (y > m) rangeApply(p << 1 | 1, x, y, m + 1, r, v);
43     pull(p);
44 }
45 void rangeApply(i32 x, i32 y, const Tag &v) {
46     rangeApply(1, x, y, 0, n - 1, v);
47 }
48 void apply(i32 x, const Tag &v) {
49     rangeApply(1, x, x, 0, n - 1, v);
50 }
51 Info rangeQuery(i32 p, i32 x, i32 y, i32 l, i32 r) {
52     if (x <= l && y >= r) return info[p];
53     Info res;
54     push(p, l, r);
55     i32 m = (l + r) / 2;
56     if (x <= m) res = res + rangeQuery(p << 1, x, y, l, m);
57     if (y > m) res = res + rangeQuery(p << 1 | 1, x, y, m + 1, r);
58     pull(p);
59     return res;
60 }
61 Info rangeQuery(i32 x, i32 y) {

```

```

62     return rangeQuery(1, x, y, 0, n - 1);
63 }
64 Info query(i32 x) {
65     return rangeQuery(1, x, x, 0, n - 1);
66 }
67 };
68
69 struct Tag {
70     i64 add = 0;
71     void apply(const Tag &v) {
72         add += v.add;
73     }
74 };
75
76 struct Info {
77     i64 sum = 0, len = 1;
78     i64 mn = INF, mx = -INF;
79     void init(const i64 &x) {
80         mn = mx = x;
81         sum = x;
82     }
83     void apply(const Tag &v) {
84         sum += len * v.add;
85         mn += v.add;
86         mx += v.add;
87     }
88     Info operator + (const Info &a) {
89         Info res;
90         res.mn = std::min(mn, a.mn);
91         res.mx = std::max(mx, a.mx);
92         res.sum = sum + a.sum;
93         res.len = len + a.len;
94         return res;
95     }
96 };

```

#### 4.6.4 PersistentSegmentTreeByEnar

```

1 template<typename Info, typename Tag>
2 struct PersistentTree {
3     struct Node {
4         i32 l = 0, r = 0;
5         Info info;
6         Tag tag;
7     };
8     #define ls(x) (node[x].l)
9     #define rs(x) (node[x].r)
10    PersistentTree(i32 n) : PersistentTree(std::vector<Info>(n + 1)) {}

```

```

11 PersistentTree(const std::vector<Info> &init) : n((i32)init.size() - 1) {
12     node.reserve(n << 3);
13     auto build = [&](auto self, i32 l, i32 r) ->i32 {
14         node.push_back(Node());
15         i32 id = node.size() - 1;
16         if(l == r) {
17             node[id].info = init[l];
18         } else {
19             i32 mid = (l + r) / 2;
20             ls(id) = self(self, l, mid);
21             rs(id) = self(self, mid + 1, r);
22             node[id].info = node[ls(id)].info + node[rs(id)].info;
23         }
24         return id;
25     };
26     root.push_back(build(build, 1, n));
27 };
28 i32 update(i32 ver, i32 pos, const Info &val) {
29     root.push_back(update(root[ver], 1, n, pos, val));
30     return root.size() - 1;
31 }
32 i32 update(i32 ver, i32 pos, const Tag &dx) {
33     root.push_back(update(root[ver], 1, n, pos, dx));
34     return root.size() - 1;
35 }
36 Info query(i32 ver, i32 pos) {
37     return rangeQuery(ver, pos, pos);
38 }
39 Info rangeQuery(i32 ver, i32 l, i32 r) {
40     return rangeQuery(root[ver], 1, n, l, r);
41 }
42 i32 update(i32 lst, i32 l, i32 r, const i32 &pos, const Info &val) {
43     node.push_back(node[lst]);
44     i32 id = node.size() - 1;
45     if(l == r) {
46         node[id].info = val;
47     } else {
48         i32 mid = (l + r) / 2;
49         if(pos <= mid) {
50             ls(id) = update(ls(lst), l, mid, pos, val);
51         } else if(pos > mid) {
52             rs(id) = update(rs(lst), mid + 1, r, pos, val);
53         }
54         node[id].info = node[ls(id)].info + node[rs(id)].info;
55     }
56     return id;
57 }
58 i32 update(i32 lst, i32 l, i32 r, const i32 &pos, const Tag &dx) {

```

```

59     node.push_back(node[lst]);
60     i32 id = node.size() - 1;
61     if(l == r) {
62         node[id].info.apply(dx);
63     } else {
64         i32 mid = (l + r) / 2;
65         if(pos <= mid) {
66             ls(id) = update(ls(lst), l, mid, pos, dx);
67         } else if(pos > mid) {
68             rs(id) = update(rs(lst), mid + 1, r, pos, dx);
69         }
70         node[id].info = node[ls(id)].info + node[rs(id)].info;
71     }
72     return id;
73 }
74 Info rangeQuery(i32 id, i32 l, i32 r, const i32 &x, const i32 &y) {
75     if(x <= l && r <= y) {
76         return node[id].info;
77     }
78     i32 mid = (l + r) / 2;
79     Info res;
80     if(x <= mid) {
81         res = res + rangeQuery(ls(id), l, mid, x, y);
82     }
83     if(y > mid) {
84         res = res + rangeQuery(rs(id), mid + 1, r, x, y);
85     }
86     return res;
87 }
88 i32 kth(i32 verl, i32 verr, i32 k) {
89     return kth(root[verl], root[verr], 1, n, k);
90 }
91 i32 kth(i32 idx, i32 idy, i32 l, i32 r, i32 k) { //静态区间第k小，不支持修
改
92     if(l >= r) return l;
93     i32 mid = (l + r) / 2;
94     i32 dx = node[ls(idy)].info.sum - node[ls(idx)].info.sum;
95     if(dx >= k) {
96         return kth(ls(idx), ls(idy), l, mid, k);
97     } else {
98         return kth(rs(idx), rs(idy), mid + 1, r, k - dx);
99     }
100 }
101 #undef ls
102 #undef rs
103 const i32 n;
104 std::vector<Node> node;
105 std::vector<i32> root;

```



```

106 };
107
108 struct Tag {
109     Tag(i32 dx = 0) : add(dx) {}
110     i32 add = 0;
111     void apply(const Tag &dx) {
112         add += dx.add;
113     }
114 };
115
116 struct Info {
117     i32 sum = 0;
118     void apply(const Tag &dx) {
119         sum += dx.add;
120     }
121 };
122
123 Info operator+(const Info &x, const Info &y) {
124     Info res;
125     res.sum = x.sum + y.sum;
126     return res;
127 }

```

## 5 数论

### 5.1 CRT

```

1 i64 exgcd(i64 a, i64 b, i64 &x, i64 &y) {
2     if (b == 0) {
3         x = 1;
4         y = 0;
5         return a;
6     }
7     i64 d = exgcd(b, a % b, y, x);
8     y -= (a / b) * x;
9     return d;
10 }
11
12 i64 crt(i32 k, std::vector<i64> a, std::vector<i64> r) {
13     i64 n = 1;
14     i64 ans = 0;
15     for (i32 i = 0; i < k; ++i) {
16         n = n * r[i];
17     }
18     for (i32 i = 0; i < k; ++i) {
19         i64 m = n / r[i];
20         i64 b, y;

```

```

21         exgcd(m, r[i], b, y);
22         ans = (ans + a[i] * m * b % n) % n;
23     }
24     return (ans % n + n) % n;
25 }

```

### 5.2 Euclidean-Like

```

1 const i32 I2 = 499122177, I6 = 166374059;
2
3 struct Euclidean {
4     i64 f, g, h;
5     Euclidean() : f(0), g(0), h(0) {}
6     Euclidean(i64 n_, i64 a_, i64 b_, i64 c_, i64 p_) {
7         Euclidean tmp = euc(n_, a_, b_, c_, p_);
8         *this = tmp;
9     }
10    Euclidean euc(i64 n, i64 a, i64 b, i64 c, i64 p) {
11        i64 ac = a / c;
12        i64 bc = b / c;
13        i64 m = (a * n + b) / c;
14        i64 n1 = n + 1;
15        i64 n21 = n * 2 + 1;
16        Euclidean d;
17        if (a == 0) {
18            d.f = bc * n1 % p;
19            d.g = bc * n % p * n1 % p * I2 % p;
20            d.h = bc * bc % p * n1 % p;
21            return d;
22        }
23        if (a >= c || b >= c) {
24            d.f = n * n1 % p * I2 % p * ac % p
25                + bc * n1 % p;
26            d.g = ac * n % p * n1 % p * n21 % p * I6 % p
27                + bc * n % p * n1 % p * I2 % p;
28            d.h = ac * ac % p * n % p * n1 % p * n21 % p * I6 % p
29                + ac * bc % p * n % p * n1 % p
30                + bc * bc % p * n1 % p;
31            d.f %= p;
32            d.g %= p;
33            d.h %= p;
34            Euclidean e = euc(n, a % c, b % c, c, p);
35            d.h += e.h + 2 * bc % p * e.f % p + 2 * ac % p * e.g % p;
36            d.g += e.g;
37            d.f += e.f;
38            d.f %= p;
39            d.g %= p;
40            d.h %= p;

```

```

41     return d;
42 }
43 Euclidean e = euc(m - 1, c, c - b - 1, a, p);
44 d.f = n * m % p - e.f;
45 d.f = (d.f % p + p) % p;
46 d.g = m * n % p * n1 % p - e.h - e.f;
47 d.g = (d.g * I2 % p + p) % p;
48 d.h = n * m % p * (m + 1) % p - 2 * e.g - 2 * e.f - d.f;
49 d.h = (d.h % p + p) % p;
50 return d;
51 }
52 };

```

### 5.3 Ex-GCD

```

1 i64 exgcd(i64 a, i64 b, i64 &x, i64 &y) {
2     if (b == 0) {
3         x = 1;
4         y = 0;
5         return a;
6     }
7     i64 d = exgcd(b, a % b, y, x);
8     y -= (a / b) * x;
9     return d;
10 }

```

### 5.4 Factorize

```

1 const i64 BASE1[7] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022};
2 const i64 BASE2[9] = {2, 3, 5, 7, 11, 13, 17, 19, 23};
3
4 i64 power(i64 a, i64 b, i64 p) {
5     i64 res = 1;
6     while (b) {
7         if (b & 1) res = (i128)res * a % p;
8         a = (i128)a * a % p;
9         b >>= 1;
10    }
11    return res;
12 }
13
14 bool millerRabin(i64 n) {
15     if (n < 3 || n % 2 == 0) return n == 2;
16     if (n % 3 == 0) return n == 3;
17     i64 u = n - 1, t = 0;
18     while (u % 2 == 0) {
19         u /= 2;

```

```

20         t++;
21     }
22     for (i32 i = 0; i < 9; ++i) {
23         i64 v = power(BASE1[i], u, n);
24         if (v == 1) continue;
25         for (i32 s = 0; s <= t; ++s) {
26             if(s == t) return false;
27             if(v == n - 1) break;
28             v = (i128)v * v % n;
29         }
30     }
31     return true;
32 }
33
34 i64 pollardRho(i64 n) {
35     static std::mt19937_64 rng(std::chrono::steady_clock::now().
36         time_since_epoch().count());
37     std::uniform_int_distribution<i64> rangeRand(1, n - 1);
38     i64 c = rangeRand(rng);
39     auto f = [&](i64 x) -> i64 {
40         return (static_cast<i128>(x) * x + c) % n;
41     };
42     i64 t = f(0), r = f(t);
43     while (t != r) {
44         i64 d = std::__gcd(std::abs(t - r), n);
45         if (d > 1) return d;
46         r = f(r);
47         t = f(t);
48     }
49     return n;
50 }
51 std::vector<i64> factorize(i64 n) {
52     std::vector<i64> p;
53     std::function<void(i64)> work = [&](i64 num) {
54         if (num <= 10000) {
55             for (i32 i = 2; i * i <= num; ++i) {
56                 while (num % i == 0) {
57                     p.push_back(i);
58                     num /= i;
59                 }
60             }
61             if (num > 1) p.push_back(num);
62             return;
63         }
64         if (millerRabin(num)) {
65             p.push_back(num);
66             return;

```

```

67     }
68     i64 x = num;
69     while (x == num) x = pollardRho(num);
70     work(num / x);
71     work(x);
72 };
73 work(n);
74 std::sort(p.begin(), p.end());
75 return p;
76 }

```

## 5.5 GCD-LCM

```

1 i64 gcd(i64 a, i64 b) {
2     return a ? gcd(b % a, a) : b;
3 }
4
5 i64 lcm(i64 a, i64 b) {
6     return a * b / gcd(a, b);
7 }

```

## 5.6 LCE

```

1 i64 exgcd(i64 a, i64 b, i64 &x, i64 &y) {
2     if (b == 0) {
3         x = 1;
4         y = 0;
5         return a;
6     }
7     i64 d = exgcd(b, a % b, y, x);
8     y -= (a / b) * x;
9     return d;
10 }
11
12 bool lieu(i64 a, i64 b, i64 c, i64 &x, i64 &y) {
13     i64 d = exgcd(a, b, x, y);
14     if (c % d != 0) return false;
15     i64 k = c / d;
16     x *= k;
17     y *= k;
18     return true;
19 }

```

## 5.7 Miller-Rabin

```

1 const i64 BASE1[7] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022};

```

```

2 const i64 BASE2[9] = {2, 3, 5, 7, 11, 13, 17, 19, 23};
3
4 i64 power(i64 a, i64 b, i64 p) {
5     i64 res = 1;
6     while (b) {
7         if (b & 1) res = (i128)res * a % p;
8         a = (i128)a * a % p;
9         b >>= 1;
10    }
11    return res;
12 }
13
14 bool millerRabin(i64 n) {
15     if (n < 3 || n % 2 == 0) return n == 2;
16     if (n % 3 == 0) return n == 3;
17     i64 u = n - 1, t = 0;
18     while (u % 2 == 0) {
19         u /= 2;
20         t++;
21     }
22     for (i32 i = 0; i < 9; ++i) {
23         i64 v = power(BASE1[i], u, n);
24         if (v == 1) continue;
25         for (i32 s = 0; s <= t; ++s) {
26             if (s == t) return false;
27             if (v == n - 1) break;
28             v = (i128)v * v % n;
29         }
30     }
31     return true;
32 }

```

## 5.8 Phi

```

1 std::vector<bool> vis;
2 std::vector<i32> phi, primes;
3
4 void getPhi(i32 n) {
5     phi.assign(n + 1, 0);
6     vis.assign(n + 1, false);
7     phi[1] = 1;
8     for (i32 i = 2; i <= n; ++i) {
9         if (!vis[i]) {
10             primes.push_back(i);
11             phi[i] = i - 1;
12         }
13         for (const auto &p : primes) {
14             i64 m = i * p;

```

```

15     if (m > n) {
16         break;
17     }
18     vis[m] = true;
19     if (i % p == 0) {
20         phi[m] = phi[i] * p;
21         break;
22     } else {
23         phi[m] = phi[i] * (p - 1);
24     }
25 }
26 }
27 }

```

## 5.9 Pollard-Rho

```

1 std::mt19937_64 rnd(std::chrono::steady_clock::now().time_since_epoch().count()
2 );
3 i64 pollardRho(i64 n) {
4     std::uniform_int_distribution<i64> rangeRand(1, n - 1);
5     i64 c = rangeRand(rnd);
6     auto f = [&](i64 x) -> i64 {
7         return (static_cast<i128>(x) * x + c) % n;
8     };
9     i64 t = f(0), r = f(t);
10    while (t != r) {
11        i64 d = std::gcd(std::abs(t - r), n);
12        if (d > 1) return d;
13        r = f(r);
14        t = f(t);
15    }
16    return n;
17 }

```

## 5.10 Sieve

```

1 std::vector<i32> minp, primes;
2
3 void sieve(i32 n) {
4     primes.clear();
5     minp.assign(n + 1, 0);
6     for(i32 i = 2; i <= n; i++) {
7         if (minp[i] == 0) {
8             minp[i] = i;
9             primes.push_back(i);
10        }

```

```

11        for (auto p : primes) {
12            if (i * p > n) {
13                break;
14            }
15            minp[i * p] = p;
16            if (p == minp[i]) {
17                break;
18            }
19        }
20    }
21 }

```

## 6 杂类

### 6.1 QuickRead

```

1 inline i64 read() {
2     char s;
3     i64 k = 0, base = 1;
4     while ((s = getchar()) != '-' && s != EOF && !(s >= '0' && s <= '9'));
5     if (s == EOF) exit(0);
6     if (s == '-') base = -1, s = getchar();
7     while (s >= '0' && s <= '9') {
8         k = k * 10 + (s - '0');
9         s = getchar();
10    }
11    return k * base;
12 }
13
14 inline void write(i64 x) {
15     if (x > 9) write(x / 10);
16     putchar(x % 10 | 48);
17 }

```

## 7 计算几何

### 7.1 BinaryHull

```

1 const double EPS = 1E-10;
2
3 i32 fcmp(double x) {
4     if (fabs(x) < EPS) return 0;
5     else if (x < 0) return -1;
6     else return 1;
7 }
8

```

```

9 struct Vecteur {
10     double x, y;
11     Vecteur() {}
12     Vecteur(double x, double y) : x(x), y(y) {}
13     Vecteur operator + (const Vecteur &u) {
14         return Vecteur(x + u.x, y + u.y);
15     }
16     Vecteur operator - (const Vecteur &u) {
17         return Vecteur(x - u.x, y - u.y);
18     }
19     Vecteur operator * (const double &k) {
20         return Vecteur(k * x, k * y);
21     }
22     Vecteur operator / (double &k) {
23         if (k < EPS) k += EPS;
24         return Vecteur(x / k, y / k);
25     }
26     bool operator < (const Vecteur &u) const {
27         return (fcmp(x - u.x) == -1) || (fcmp(x - u.x) == 0 && fcmp(y - u.y) ==
28             -1);
29     };
30     double dot(Vecteur a, Vecteur b) {
31         return a.x * b.x + a.y * b.y;
32     }
33     double cross(Vecteur a, Vecteur b) {
34         return a.x * b.y - a.y * b.x;
35     }
36     using Point = Vecteur;
37
38     auto binaryHull(std::vector<Point> p) {
39         std::sort(p.begin(), p.end());
40         std::vector<Point> up, dn;
41         for (auto &u : p) {
42             while (up.size() > 1 && cross(up.back() - up[up.size() - 2], u - up.
43                 back()) >= 0) {
44                 up.pop_back();
45             }
46             while (!up.empty() && up.back().x == u.x) {
47                 up.pop_back();
48             }
49             up.push_back(u);
50             while (dn.size() > 1 && cross(dn.back() - dn[dn.size() - 2], u - dn.
51                 back()) <= 0) {
52                 dn.pop_back();
53             }
54             if (dn.empty() || dn.back().x < u.x) {
55                 dn.push_back(u);

```

```

54     }
55 }
56 return std::make_pair(up, dn);
57 }

```

## 7.2 ConvexHull

```

1 const double EPS = 1E-10;
2
3 i32 fcmp(double x) {
4     if (fabs(x) < EPS) return 0;
5     else if (x < 0) return -1;
6     else return 1;
7 }
8
9 struct Vecteur {
10     double x, y;
11     Vecteur() {}
12     Vecteur(double x, double y) : x(x), y(y) {}
13     Vecteur operator + (const Vecteur &u) {
14         return Vecteur(x + u.x, y + u.y);
15     }
16     Vecteur operator - (const Vecteur &u) {
17         return Vecteur(x - u.x, y - u.y);
18     }
19     Vecteur operator * (const double &k) {
20         return Vecteur(k * x, k * y);
21     }
22     Vecteur operator / (double &k) {
23         if (k < EPS) k += EPS;
24         return Vecteur(x / k, y / k);
25     }
26     bool operator < (const Vecteur &u) const {
27         return (fcmp(x - u.x) == -1) || (fcmp(x - u.x) == 0 && fcmp(y - u.y) ==
28             -1);
29     };
30     double dot(Vecteur a, Vecteur b) {
31         return a.x * b.x + a.y * b.y;
32     }
33     double cross(Vecteur a, Vecteur b) {
34         return a.x * b.y - a.y * b.x;
35     }
36     using Point = Vecteur;
37
38     auto convexHull(std::vector<Point> p) {
39         std::sort(p.begin(), p.end());
40         std::vector<Point> res;

```

```

41 i32 n = p.size();
42 for (i32 i = 0; i < n; ++i) {
43     while (res.size() > 1 && cross(res.back() - res[res.size() - 2], p[i] -
44         res.back()) <= 0) {
45         res.pop_back();
46     }
47     res.push_back(p[i]);
48 }
49 i32 m = res.size();
50 for (i32 i = n - 2; i >= 0; --i) {
51     while (res.size() > m && cross(res.back() - res[res.size() - 2], p[i] -
52         res.back()) <= 0) {
53         res.pop_back();
54     }
55     res.push_back(p[i]);
56 }
57 if (res.size() > 1) res.pop_back();
58 return res;
59 }

```

### 7.3 Line

```

1 const double EPS = 1E-10;
2
3 i32 fcmp(double x) {
4     if (fabs(x) < EPS) return 0;
5     else if (x < 0) return -1;
6     else return 1;
7 }
8
9 struct Vecteur {
10     double x, y;
11     Vecteur() = default;
12     Vecteur(double x, double y) : x(x), y(y) {}
13     Vecteur(Vecteur a, Vecteur b) : x(b.x - a.x), y(b.y - a.y) {}
14     Vecteur operator + (const Vecteur &u) const {
15         return {x + u.x, y + u.y};
16     }
17     Vecteur operator - (const Vecteur &u) const {
18         return {x - u.x, y - u.y};
19     }
20     Vecteur operator * (const double &k) const {
21         return {k * x, k * y};
22     }
23     Vecteur operator / (double k) const {
24         if (k < EPS) k += EPS;
25         return {x / k, y / k};
26     }

```

```

27 bool operator < (const Vecteur &u) const {
28     return (fcmp(x - u.x) == -1) || (fcmp(x - u.x) == 0 && fcmp(y - u.y) ==
29         -1);
30 }
31 bool operator == (const Vecteur &u) const {
32     return (fcmp(x - u.x) == 0) && (fcmp(y - u.y) == 0);
33 }
34 double abs2() const {
35     return x * x + y * y;
36 }
37 double abs() const {
38     return std::sqrt(abs2());
39 }
40 double arg() const {
41     return atan2(y, x);
42 }
43 Vecteur rotate(double rad) {
44     return {x * cos(rad) - y * sin(rad), x * sin(rad) + y * cos(rad)};
45 }
46 Vecteur unit() {
47     double len = abs();
48     if (fcmp(len) == 0) {
49         return Vecteur(0, 0);
50     } else {
51         return Vecteur(-y / len, x / len);
52     }
53 }
54 Vecteur norm() {
55     return {-y, x};
56 }
57 double dot(Vecteur a, Vecteur b) {
58     return a.x * b.x + a.y * b.y;
59 }
60 double cross(Vecteur a, Vecteur b) {
61     return a.x * b.y - a.y * b.x;
62 }
63 using Point = Vecteur;
64
65 struct Line : Vecteur {
66     Point p;
67     Line() {}
68     Line(Point p, Vecteur d) : Vecteur(d), p(p) {}
69     Line(double k, double b) : Vecteur{1, k}, p{0, b} {}
70     Line(double a, double b, double c) : Vecteur{-b, a}, p{0, -c / b} {}
71     double operator () (const Vecteur &u) const {
72         return dot(u, Vecteur(*this).norm()) + p.y * x;
73     }

```

```

74     bool operator < (const Line &u) const {
75         i32 rst = fcmp(Vecteur(*this).arg() - u.arg());
76         if (rst == 0) {
77             return (u.p.x - p.x) * (u.p.y - p.x) > EPS;
78         } else {
79             return rst < 0;
80         }
81     }
82 };
83 Point junct(Line a, Line b) {
84     Vecteur u = a.p - b.p;
85     double t = cross(Vecteur(b), u) / cross(Vecteur(a), Vecteur(b));
86     return a.p + Vecteur(a) * t;
87 }
88 Vecteur refl(Vecteur v, Line l) {
89     return v - Vecteur(l) * (l(v) / l.abs2() * 2);
90 }
91 Vecteur proj(Vecteur v, Line l) {
92     return v - Vecteur(l) * (l(v) / l.abs2());
93 }
94 double dist(Point p, Line l) {
95     return l(p) / p.abs();
96 }
97 bool isPara(Line a, Line b) {
98     return !fcmp(cross(a, b));
99 }
100 bool isVert(Line a, Line b) {
101     return !fcmp(dot(a, b));
102 }
103 bool isSdrt(Line a, Line b) {
104     return isPara(a, b) && dot(a, b) > 0;
105 }
106 bool online(Point p, Line l) {
107     return !fcmp(l(p));
108 }

```

## 7.4 RotatingCalipers

```

1  const double EPS = 1E-10;
2
3  i32 fcmp(double x) {
4      if (fabs(x) < EPS) return 0;
5      else if (x < 0) return -1;
6      else return 1;
7  }
8
9  struct Vecteur {
10     double x, y;

```

```

11     Vecteur() {}
12     Vecteur(double x, double y) : x(x), y(y) {}
13     Vecteur(Vecteur a, Vecteur b) : x(b.x - a.x), y(b.y - a.y) {}
14     Vecteur operator + (const Vecteur &u) {
15         return Vecteur(x + u.x, y + u.y);
16     }
17     Vecteur operator - (const Vecteur &u) {
18         return Vecteur(x - u.x, y - u.y);
19     }
20     Vecteur operator * (const double &k) {
21         return Vecteur(k * x, k * y);
22     }
23     Vecteur operator / (double &k) {
24         if (k < EPS) k += EPS;
25         return Vecteur(x / k, y / k);
26     }
27     bool operator < (const Vecteur &u) const {
28         return (fcmp(x - u.x) == -1) || (fcmp(x - u.x) == 0 && fcmp(y - u.y) ==
29             -1);
30     }
31     bool operator == (const Vecteur &u) const {
32         return (fcmp(x - u.x) == 0) && (fcmp(y - u.y) == 0);
33     }
34     bool operator != (const Vecteur &u) const {
35         return (fcmp(x - u.x) != 0) || (fcmp(y - u.y) != 0);
36     }
37     double dot(Vecteur a, Vecteur b) {
38         return a.x * b.x + a.y * b.y;
39     }
40     double cross(Vecteur a, Vecteur b) {
41         return a.x * b.y - a.y * b.x;
42     }
43     double dist2(Vecteur a, Vecteur b) {
44         return std::pow(a.x - b.x, 2) + std::pow(a.y - b.y, 2);
45     }
46     double dist(Vecteur a, Vecteur b) {
47         return std::sqrt(dist2(a, b));
48     }
49     using Point = Vecteur;
50
51     auto convexHull(std::vector<Point> p) {
52         std::sort(p.begin(), p.end());
53         std::vector<Point> res;
54         i32 n = p.size();
55         for (i32 i = 0; i < n; ++i) {
56             while (res.size() > 1 && cross(res.back() - res[res.size() - 2], p[i] -
                    res.back()) <= 0) {

```

```

57     res.pop_back();
58 }
59 res.push_back(p[i]);
60 }
61 i32 m = res.size();
62 for (i32 i = n - 2; i >= 0; --i) {
63     while (res.size() > m && cross(res.back() - res[res.size() - 2], p[i] -
64         res.back()) <= 0) {
65         res.pop_back();
66     }
67     res.push_back(p[i]);
68 }
69 if (res.size() > 1) res.pop_back();
70 return res;
71 }
72 double rotateCalipers(std::vector<Point> h) {
73     h.push_back(h.front());
74     i32 n = h.size();
75     if (n < 4) {
76         return dist2(h[0], h[1]);
77     }
78     double res = 0;
79     for (i32 i = 1, j = 1; i < n; ++i) {
80         Vecteur b(h[i - 1], h[i]);
81         while (cross(b, Vecteur(h[i - 1], h[j])) <= cross(b, Vecteur(h[i - 1],
82             h[(j + 1) % n]))) {
83             j = (j + 1) % n;
84         }
85         res = std::max({res, dist2(h[i - 1], h[j]), dist2(h[i], h[j])});
86     }
87     return res;
88 }

```

## 7.5 Segment

```

1 const double EPS = 1E-10;
2
3 i32 fcmp(double x) {
4     if (fabs(x) < EPS) return 0;
5     else if (x < 0) return -1;
6     else return 1;
7 }
8
9 struct Vecteur {
10     double x, y;
11     Vecteur() {}
12     Vecteur(double x, double y) : x(x), y(y) {}

```

```

13 Vecteur operator + (Vecteur &u) {
14     return Vecteur(x + u.x, y + u.y);
15 }
16 Vecteur operator - (Vecteur &u) {
17     return Vecteur(x - u.x, y - u.y);
18 }
19 Vecteur operator * (double &k) {
20     return Vecteur(k * x, k * y);
21 }
22 Vecteur operator / (double &k) {
23     if (k < EPS) k += EPS;
24     return Vecteur(x / k, y / k);
25 }
26 bool operator < (const Vecteur &u) const {
27     return (fcmp(x - u.x) == -1) || (fcmp(x - u.x) == 0 && fcmp(y - u.y) ==
28         -1);
29 }
30 bool operator == (const Vecteur &u) const {
31     return (fcmp(x - u.x) == 0) && (fcmp(y - u.y) == 0);
32 }
33 double arg() {
34     return atan2(y, x);
35 }
36 double abs() {
37     return std::sqrt(x * x + y * y);
38 }
39 Vecteur unit() {
40     double len = abs();
41     if (fcmp(len) == 0) return Vecteur(0, 0);
42     return Vecteur(-y / len, x / len);
43 }
44 Vecteur rotate(double rad) {
45     return Vecteur(x * cos(rad) - y * sin(rad), x * sin(rad) + y * cos(rad)
46 );
47 };
48 double dot(Vecteur a, Vecteur b) {
49     return a.x * b.x + a.y * b.y;
50 }
51 double cross(Vecteur a, Vecteur b) {
52     return a.x * b.y - a.y * b.x;
53 }
54 double cosine(Vecteur a, Vecteur b) {
55     return dot(a, b) / a.abs() / b.abs();
56 }
57 double arg(Vecteur a, Vecteur b) {
58     return acos(cosine(a, b));
59 }

```



```

59 double area(Vecteur a, Vecteur b, Vecteur c) {
60     return cross(b - a, c - a) / 2.0;
61 }
62 using Point = Vecteur;
63
64 i32 ccw(Point a, Point b, Point c) {
65     i32 sign = fcmp((b - a) * (c - a));
66     if (sign == 0) {
67         if (fcmp((b - a) % (c - a)) == -1) return 2;
68         if ((c - a).norm() > (b - a).norm() + EPS) return -2;
69     }
70     return sign;
71 }
72
73 struct Segment {
74     Point x, y;
75     Segment() {}
76     Segment(Point x, Point y) : x(x), y(y) {}
77     bool isCross(const Point &p) {
78         return (p - x) % (p - y) < EPS && std::fabs((p - x) * (p - y)) < EPS;
79     }
80     bool isCross(const Segment &s) {
81         return ccw(x.x, x.y, y.x) * ccw(x.x, x.y, y.y) <= 0
82             && ccw(y.x, y.y, x.x) * ccw(y.x, y.y, x.y) <= 0;
83     }
84 };

```

## 7.6 Vecteur

```

1  const double EPS = 1E-10;
2
3  i32 fcmp(double x) {
4      if (fabs(x) < EPS) return 0;
5      else if (x < 0) return -1;
6      else return 1;
7  }
8
9  struct Vecteur {
10     double x, y;
11     Vecteur() {}
12     Vecteur(double x, double y) : x(x), y(y) {}
13     Vecteur(Vecteur a, Vecteur b) : x(b.x - a.x), y(b.y - a.y) {}
14     Vecteur operator + (Vecteur &u) {
15         return Vecteur(x + u.x, y + u.y);
16     }
17     Vecteur operator - (Vecteur &u) {
18         return Vecteur(x - u.x, y - u.y);
19     }

```

```

20     Vecteur operator * (double &k) {
21         return Vecteur(k * x, k * y);
22     }
23     Vecteur operator / (double &k) {
24         if (k < EPS) k += EPS;
25         return Vecteur(x / k, y / k);
26     }
27     bool operator < (const Vecteur &u) const {
28         return (fcmp(x - u.x) == -1) || (fcmp(x - u.x) == 0 && fcmp(y - u.y) ==
29             -1);
30     }
31     bool operator == (const Vecteur &u) const {
32         return (fcmp(x - u.x) == 0) && (fcmp(y - u.y) == 0);
33     }
34     double arg() {
35         return atan2(y, x);
36     }
37     double abs() {
38         return std::sqrt(x * x + y * y);
39     }
40     Vecteur rotate(double rad) {
41         return Vecteur(x * cos(rad) - y * sin(rad), x * sin(rad) + y * cos(rad)
42             );
43     }
44     Vecteur unit() {
45         double len = abs();
46         if (fcmp(len) == 0) {
47             return Vecteur(0, 0);
48         } else {
49             return Vecteur(-y / len, x / len);
50         }
51     }
52     Vecteur norm() {
53         return {-y, x};
54     }
55 };
56
57 double dot(Vecteur a, Vecteur b) {
58     return a.x * b.x + a.y * b.y;
59 }
60
61 double cross(Vecteur a, Vecteur b) {
62     return a.x * b.y - a.y * b.x;
63 }
64
65 double cosine(Vecteur a, Vecteur b) {
66     return dot(a, b) / a.abs() / b.abs();
67 }
68
69 double arg(Vecteur a, Vecteur b) {
70     return acos(cosine(a, b));
71 }

```

```
66 double area(Vecteur a, Vecteur b, Vecteur c) {  
67     return cross(b - a, c - a) / 2.0;  
68 }  
69 double dist2(Vecteur a, Vecteur b) {  
70     return std::pow(a.x - b.x, 2) + std::pow(a.y - b.y, 2);  
71 }  
72 double dist(Vecteur a, Vecteur b) {  
73     return std::sqrt(dist2(a, b));  
74 }  
75 using Point = Vecteur;
```