



## Notebook - Maratona de Programação

Cabo HDMI, VGA, USB

### Contents

<b>1</b>	<b>Graph</b>	<b>2</b>
1.1	Dfs tree . . . . .	2
1.2	Bellman Ford . . . . .	2
1.3	2SAT . . . . .	2
1.4	Dominator tree . . . . .	3
1.5	Kth Ancestor . . . . .	4
1.6	Topological Sort . . . . .	4
1.7	Dijkstra . . . . .	4
1.8	Dinic Min cost . . . . .	4
1.9	Dinic . . . . .	6
<b>2</b>	<b>Segtree</b>	<b>7</b>
2.1	Standard SegTree . . . . .	7
2.2	Persistent sum segment tree . . . . .	7
2.3	Set and update lazy seg . . . . .	8
2.4	Lazy SegTree . . . . .	8
<b>3</b>	<b>Set</b>	<b>9</b>
3.1	Set . . . . .	9
3.2	Multiset . . . . .	9
3.3	Ordered Set . . . . .	9
<b>4</b>	<b>Misc</b>	<b>10</b>
4.1	Template . . . . .	10
<b>5</b>	<b>String</b>	<b>10</b>
5.1	String hash . . . . .	10
5.2	Suffix Array . . . . .	10
5.3	Z function . . . . .	11
<b>6</b>	<b>Algorithms</b>	<b>11</b>
6.1	Sparse table . . . . .	11
<b>7</b>	<b>Parser</b>	<b>11</b>
7.1	Parsing Functions . . . . .	11

# 1 Graph

## 1.1 Dfs tree

Complexity:  $O(E + V)$

```
1 // TITLE: Dfs tree
2 // COMPLEXITY:  $O(E + V)$ 
3 // DESCRIPTION: Create dfs tree from graph
4
5 int desce[mxN], sobe[mxN];
6 int backedges[mxN], vis[mxN];
7 int pai[mxN], h[mxN];
8
9 void dfs(int a, int p) {
10     if(vis[a]) return;
11     pai[a] = p;
12     h[a] = h[p]+1;
13     vis[a] = 1;
14
15     for(auto b : g[a]) {
16         if (p == b) continue;
17         if (vis[b]) continue;
18         dfs(b, a);
19         backedges[a] += backedges[b];
20     }
21     for(auto b : g[a]) {
22         if(h[b] > h[a]+1)
23             desce[a]++;
24         else if(h[b] < h[a]-1)
25             sobe[a]++;
26     }
27     backedges[a] += sobe[a] - desce[a];
28 }
```

## 1.2 Bellman Ford

Complexity:  $O(n * m)$  |  $n = |\text{nodes}|$ ,  $m = |\text{edges}|$

Finds shortest paths from a starting node to all nodes of the graph. Detects negative cycles, if they exist.

```
1 // TITLE: Bellman Ford
2 // COMPLEXITY:  $O(n * m)$  |  $n = |\text{nodes}|$ ,  $m = |\text{edges}|$ 
3 // DESCRIPTION: Finds shortest paths from a starting
4 // node to all nodes of the graph. Detects negative
5 // cycles, if they exist.
6
7 // a and b vertices, c cost
8 // [{a, b, c}, {a, b, c}]
9 vector<tuple<int, int, int>> edges;
10 int N;
11
12 void bellman_ford(int x){
13     for (int i = 0; i < N; i++){
14         dist[i] = oo;
15     }
16     dist[x] = 0;
17
18     for (int i = 0; i < N - 1; i++){
19         for (auto [a, b, c]: edges){
20             if (dist[a] == oo) continue;
21             dist[b]= min(dist[b], dist[a] + w);
22         }
23     }
24     // return true if has cycle
25     bool check_negative_cycle(int x){
26         for (int i = 0; i < N; i++){
27             dist[i] = oo;
```

```
28     dist[x] = 0;
29
30     for (int i = 0; i < N - 1; i++){
31         for (auto [a, b, c]: edges){
32             if (dist[a] == oo) continue;
33             dist[b]= min(dist[b], dist[a] + w);
34         }
35     }
36
37     for (auto [a, b, c]: edges){
38         if (dist[a] == oo) continue;
39         if (dist[a] + w < dist[b]){
40             return true;
41         }
42     }
43     return false;
44 }
45 '''
```

## 1.3 2SAT

Complexity:  $O(n+m)$ ,  $n$  = number of variables,  $m$  = number of conjunctions (ands).

Finds an assignment that makes a certain boolean formula true, or determines that such an assignment does not exist.

```
1 // TITLE: 2SAT
2 // COMPLEXITY:  $O(n+m)$ ,  $n$  = number of variables,  $m$  =
3 // number of conjunctions (ands).
4 // DESCRIPTION: Finds an assignment that makes a
5 // certain boolean formula true, or determines that
6 // such an assignment does not exist.
7
8 struct twosat {
9     vi vis, degin;
10     stack<int> tout;
11     vector<vi> g, gi, con, sccg;
12     vi repr, conv;
13     int gsize;
14     void dfs1(int a) {
15         if (vis[a]) return;
16         vis[a]=true;
17
18         for(auto& b : g[a]) {
19             dfs1(b);
20         }
21
22         tout.push(a);
23     }
24
25     void dfs2(int a, int orig) {
26         if (vis[a]) return;
27         vis[a]=true;
28
29         repr[a]=orig;
30         sccg[orig].pb(a);
31         for(auto& b : gi[a]) {
32             if (vis[b]) {
33                 if (repr[b] != orig) {
34                     con[repr[b]].pb(orig);
35                     degin[orig]++;
36                 }
37                 continue;
38             }
39             dfs2(b, orig);
40         }
41     }
42
43     // if s1 = 1 and s2 = 1 this adds a \ b to the
44     // graph
45     void addedge(int a, int s1,
```

```

42         int b, int s2) {
43     g[2*a+(!s1)].pb(2*b+s2);
44     gi[2*b+s2].pb(2*a+(!s1));
45
46     g[2*b+(!s2)].pb(2*a+s1);
47     gi[2*a+s1].pb(2*b+(!s2));
48 }
49
50
51 twosat(int nvars) {
52     gsize=2*nvars;
53     g.assign(gsize, vi());
54     gi.assign(gsize, vi());
55     con.assign(gsize, vi());
56     sccg.assign(gsize, vi());
57     repr.assign(gsize, -1);
58     vis.assign(gsize, 0);
59     degin.assign(gsize, 0);
60 }
61
62 // returns empty vector if the formula is not
63 // satisfiable.
64 vi run() {
65     vi vals(gsize/2, -1);
66     rep(i,0,gsize) dfs1(i);
67     vis.assign(gsize,0);
68     while(!tout.empty()) {
69         int cur = tout.top();tout.pop();
70         if (vis[cur]) continue;
71         dfs2(cur,cur);
72         conv.pb(cur);
73     }
74
75     rep(i, 0, gsize/2) {
76         if (repr[2*i] == repr[2*i+1]) {
77             return {};
78         }
79     }
80
81     queue<int> q;
82     for(auto& v : conv) {
83         if (degin[v] == 0) q.push(v);
84     }
85
86     while(!q.empty()) {
87         int cur=q.front(); q.pop();
88         for(auto guy : sccg[cur]) {
89             int s = guy%2;
90             int idx = guy/2;
91             if (vals[idx] != -1) continue;
92             if (s) {
93                 vals[idx] = false;
94             } else {
95                 vals[idx]=true;
96             }
97         }
98         for (auto& b : con[cur]) {
99             if(--degin[b] == 0) q.push(b);
100         }
101     }
102     return vals;
103 }
104 };

```

## 1.4 Dominator tree

Complexity:  $O(E + V)$

```

1 // TITLE: Dominator tree
2 // COMPLEXITY:  $O(E + V)$ 

```

```

3 // DESCRIPTION: Builds dominator tree
4
5 vector<int> g[mxN];
6 vector<int> S, gt[mxN], T[mxN];
7 int dsu[mxN], label[mxN];
8 int sdom[mxN], idom[mxN], id[mxN];
9 int dfs_time = 0;
10
11 vector<int> bucket[mxN];
12 vector<int> down[mxN];
13
14 void prep(int a)
15 {
16     S.pb(a);
17     id[a] = ++dfs_time;
18     label[a] = sdom[a] = dsu[a] = a;
19
20     for (auto b: g[a]) {
21         if (!id[b]) {
22             prep(b);
23             down[a].pb(b);
24         }
25         gt[b].pb(a);
26     }
27 }
28
29 int fnd(int a, int flag = 0)
30 {
31     if (a == dsu[a]) return a;
32     int p = fnd(dsu[a], 1);
33     int b = label[ dsu[a] ];
34     if (id [ sdom[b] ] < id[ sdom[ label[a] ] ]) {
35         label[a] = b;
36     }
37     dsu[a] = p;
38     return (flag ? p: label[a]);
39 }
40
41 void build_dominator_tree(int root)
42 {
43     prep(root);
44     reverse(all(S));
45
46     int w;
47     for (int a: S) {
48         for (int b: gt[a]) {
49             w = fnd(b);
50             if (id[ sdom[w] ] < id[ sdom[a] ]) {
51                 sdom[a] = sdom[w];
52             }
53         }
54         gt[a].clear();
55         if (a != root) {
56             bucket[ sdom[a] ].pb(a);
57         }
58         for (int b: bucket[a]) {
59             w = fnd(b);
60             if (sdom[w] == sdom[b]) {
61                 idom[b] = sdom[b];
62             }
63             else {
64                 idom[b] = w;
65             }
66         }
67         bucket[a].clear();
68         for (int b: down[a]) {
69             dsu[b] = a;
70         }
71         down[a].clear();
72     }
73     reverse(all(S));
74     for (int a: S) {
75         if (a != root) {

```

```

76         if (idom[a] != sdom[a]) {
77             idom[a] = idom[ idom[a] ];
78         }
79         T[ idom[a] ].pb(a);
80     }
81 }
82 S.clear();
83 }

```

## 1.5 Kth Ancestor

Complexity:  $O(n * \log(n))$

Preprocess, then find in  $\log n$

```

1 // TITLE: Kth Ancestor
2 // COMPLEXITY:  $O(n * \log(n))$ 
3 // DESCRIPTION: Preprocess, then find in  $\log n$ 
4
5 const int LOG_N = 30;
6 int get_kth_ancestor(vector<vector<int>> & up, int v,
7     int k)
8 {
9     for (int j = 0; j < LOG_N; j++) {
10         if (k & ((int)1 << j)) {
11             v = up[v][j];
12         }
13     }
14     return v;
15 }
16 void solve()
17 {
18     vector<vector<int>> up(n, vector<int>(LOG_N));
19
20     for (int i = 0; i < n; i++) {
21         up[i][0] = parents[i];
22         for (int j = 1; j < LOG_N; j++) {
23             up[i][j] = up[up[i][j-1]][j-1];
24         }
25     }
26     cout << get_kth_ancestor(up, x, k) << endl;
27 }
28 }

```

## 1.6 Topological Sort

Complexity:  $O(N + M)$ , N: Vertices, M: Arestas

Retorna no do grapho em ordem topologica, se a quantidade de nos retornada nao for igual a quantidade de nos e impossivel

```

1 // TITLE: Topological Sort
2 // COMPLEXITY:  $O(N + M)$ , N: Vertices, M: Arestas
3 // DESCRIPTION: Retorna no do grapho em ordem
4     topologica, se a quantidade de nos retornada nao
5     for igual a quantidade de nos e impossivel
6
7 typedef vector<vector<int>> Adj_List;
8 typedef vector<int> Indegree_List; // How many nodes
9     depend on him
10 typedef vector<int> Order_List;    // The order in
11     which the nodes appears
12
13 Order_List kahn(Adj_List adj, Indegree_List indegree)
14 {
15     queue<int> q;
16     // priority_queue<int> q; // If you want in
17     // lexicographic order
18     for (int i = 0; i < indegree.size(); i++) {
19         if (indegree[i] == 0)
20             q.push(i);
21     }
22 }

```

```

16     }
17     vector<int> order;
18
19     while (not q.empty()) {
20         auto a = q.front();
21         q.pop();
22
23         order.push_back(a);
24         for (auto b: adj[a]) {
25             indegree[b]--;
26             if (indegree[b] == 0)
27                 q.push(b);
28         }
29     }
30     return order;
31 }
32
33 int32_t main()
34 {
35     Order_List = kahn(adj, indegree);
36     if (Order_List.size() != N) {
37         cout << "IMPOSSIBLE" << endl;
38     }
39     return 0;
40 }
41 }

```

## 1.7 Dkistra

Complexity:  $O(E + V \cdot \log(V))$

```

1 // TITLE: Dkistra
2 // COMPLEXITY:  $O(E + V \cdot \log(V))$ 
3 // DESCRIPTION: Finds to shortest path from start
4
5 int dist[mxN];
6 bool vis[mxN];
7 vector<pair<int, int>> g[mxN];
8
9 void dikstra(int start)
10 {
11     fill(dist, dist + mxN, oo);
12     fill(vis, vis + mxN, 0);
13     priority_queue<pair<int, int>> q;
14     dist[start] = 0;
15     q.push({0, start});
16
17     while(!q.empty()) {
18         auto [d, a] = q.top();
19         q.pop();
20         if (vis[a]) continue;
21         vis[a] = true;
22         for (auto [b, w]: g[a]) {
23             if (dist[a] + w < dist[b]) {
24                 dist[b] = dist[a] + w;
25                 q.push({-dist[b], b});
26             }
27         }
28     }
29 }

```

## 1.8 Dinic Min cost

Complexity:  $O(V \cdot V \cdot E)$ , Bipartite is  $O(\sqrt{V} \cdot E)$

Gives you the max\_flow with the min cost

```

1 // TITLE: Dinic Min cost
2 // COMPLEXITY:  $O(V \cdot V \cdot E)$ , Bipartite is  $O(\sqrt{V} \cdot E)$ 

```

```

3 // DESCRIPTION: Gives you the max_flow with the min
  cost
4
5 // Edge structure
6 struct Edge
7 {
8     int from, to;
9     int flow, capacity;
10    int cost;
11
12    Edge(int from_, int to_, int flow_, int capacity_
13        , int cost_)
14        : from(from_), to(to_), flow(flow_), capacity
15        (capacity_), cost(cost_)
16    {}
17 };
18
19 struct Dinic
20 {
21     vector<vector<int>> graph;
22     vector<Edge> edges;
23     vector<int> dist;
24     vector<bool> inqueue;
25     int size;
26     int cost = 0;
27
28     Dinic(int n)
29     {
30         graph.resize(n);
31         dist.resize(n);
32         inqueue.resize(n);
33         size = n;
34         edges.clear();
35
36     void add_edge(int from, int to, int capacity, int
37         cost)
38     {
39         edges.emplace_back(from, to, 0, capacity,
40         cost);
41         graph[from].push_back(edges.size() - 1);
42
43         edges.emplace_back(to, from, 0, 0, -cost);
44         graph[to].push_back(edges.size() - 1);
45     }
46
47     int get_max_flow(int source, int sink)
48     {
49         int max_flow = 0;
50         vector<int> next(size);
51         while(spfa(source, sink)) {
52             next.assign(size, 0);
53             for (int f = dfs(source, sink, next, oo);
54                 f != 0; f = dfs(source, sink, next, oo)) {
55                 max_flow += f;
56             }
57         }
58         return max_flow;
59     }
60
61     bool spfa(int source, int sink)
62     {
63         dist.assign(size, oo);
64         inqueue.assign(size, false);
65         queue<int> q;
66         q.push(source);
67         dist[source] = 0;
68         inqueue[source] = true;
69
70         while(!q.empty()) {
71             int a = q.front();
72             q.pop();
73             inqueue[a] = false;
74
75             for (int & b: graph[a]) {
76                 auto edge = edges[b];
77                 int cap = edge.capacity - edge.flow;
78                 if (cap > 0 && dist[edge.to] > dist[
79                     edge.from] + edge.cost) {
80                     dist[edge.to] = dist[edge.from] +
81                     edge.cost;
82                     if (not inqueue[edge.to]) {
83                         q.push(edge.to);
84                         inqueue[edge.to] = true;
85                     }
86                 }
87             }
88             return dist[sink] != oo;
89         }
90
91     int dfs(int curr, int sink, vector<int> & next,
92         int flow)
93     {
94         if (curr == sink) return flow;
95         int num_edges = graph[curr].size();
96
97         for (; next[curr] < num_edges; next[curr]++)
98         {
99             int b = graph[curr][next[curr]];
100             auto & edge = edges[b];
101             auto & rev_edge = edges[b^1];
102
103             int cap = edge.capacity - edge.flow;
104             if (cap > 0 && (dist[edge.from] + edge.
105                 cost == dist[edge.to])) {
106                 int bottle_neck = dfs(edge.to, sink,
107                     next, min(flow, cap));
108                 if (bottle_neck > 0) {
109                     edge.flow += bottle_neck;
110                     rev_edge.flow -= bottle_neck;
111                     cost += edge.cost * bottle_neck;
112                     return bottle_neck;
113                 }
114             }
115             return 0;
116         }
117     }
118
119     vector<pair<int, int>> mincut(int source, int
120         sink)
121     {
122         vector<pair<int, int>> cut;
123         spfa(source, sink);
124         for (auto & e: edges) {
125             if (e.flow == e.capacity && dist[e.from]
126                 != oo && level[e.to] == oo && e.capacity > 0) {
127                 cut.emplace_back(e.from, e.to);
128             }
129         }
130         return cut;
131     }
132 };
133
134 // Example on how to use
135 void solve()
136 {
137     int N = 10;
138
139     int source = 8;
140     int sink = 9;
141
142     Dinic flow(N);
143     flow.add_edge(8, 0, 4, 0);
144     flow.add_edge(8, 1, 3, 0);

```

```

135     flow.add_edge(8, 2, 2, 0);
136     flow.add_edge(8, 3, 1, 0);
137
138     flow.add_edge(0, 6, oo, 3);
139     flow.add_edge(0, 7, oo, 2);
140     flow.add_edge(0, 5, oo, 0);
141
142     flow.add_edge(1, 4, oo, 0);
143
144     flow.add_edge(4, 9, oo, 0);
145     flow.add_edge(5, 9, oo, 0);
146     flow.add_edge(6, 9, oo, 0);
147     flow.add_edge(7, 9, oo, 0);
148
149     int ans = flow.get_max_flow(source, sink);
150     debug(ans);
151     debug(flow.cost);
152 }
153
154 int32_t main()
155 {
156     solve();
157 }

```

## 1.9 Dinic

Complexity:  $O(V \cdot V \cdot E)$ , Bipartite is  $O(\sqrt{V} \cdot E)$

Dinic

```

1 // TITLE: Dinic
2 // COMPLEXITY:  $O(V \cdot V \cdot E)$ , Bipartite is  $O(\sqrt{V} \cdot E)$ 
3 // DESCRIPTION: Dinic
4
5 const int oo = 0x3f3f3f3f3f3f3f3f;
6 // Edge structure
7 struct Edge
8 {
9     int from, to;
10    int flow, capacity;
11
12    Edge(int from_, int to_, int flow_, int capacity_)
13        : from(from_), to(to_), flow(flow_), capacity_
14        (capacity_)
15    {}
16 };
17 struct Dinic
18 {
19     vector<vector<int>> graph;
20     vector<Edge> edges;
21     vector<int> level;
22     int size;
23
24     Dinic(int n)
25     {
26         graph.resize(n);
27         level.resize(n);
28         size = n;
29         edges.clear();
30     }
31
32     void add_edge(int from, int to, int capacity)
33     {
34         edges.emplace_back(from, to, 0, capacity);
35         graph[from].push_back(edges.size() - 1);
36
37         edges.emplace_back(to, from, 0, 0);
38         graph[to].push_back(edges.size() - 1);
39     }
40
41     int get_max_flow(int source, int sink)

```

```

42     {
43         int max_flow = 0;
44         vector<int> next(size);
45         while(bfs(source, sink)) {
46             next.assign(size, 0);
47             for (int f = dfs(source, sink, next, oo);
48                  f != 0; f = dfs(source, sink, next, oo)) {
49                 max_flow += f;
50             }
51             return max_flow;
52         }
53     }
54
55     bool bfs(int source, int sink)
56     {
57         level.assign(size, -1);
58         queue<int> q;
59         q.push(source);
60         level[source] = 0;
61
62         while(!q.empty()) {
63             int a = q.front();
64             q.pop();
65
66             for (int & b: graph[a]) {
67                 auto edge = edges[b];
68                 int cap = edge.capacity - edge.flow;
69                 if (cap > 0 && level[edge.to] == -1)
70                 {
71                     level[edge.to] = level[a] + 1;
72                     q.push(edge.to);
73                 }
74             }
75             return level[sink] != -1;
76         }
77
78         int dfs(int curr, int sink, vector<int> & next,
79                 int flow)
80         {
81             if (curr == sink) return flow;
82             int num_edges = graph[curr].size();
83
84             for (; next[curr] < num_edges; next[curr]++)
85             {
86                 int b = graph[curr][next[curr]];
87                 auto & edge = edges[b];
88                 auto & rev_edge = edges[b^1];
89
90                 int cap = edge.capacity - edge.flow;
91                 if (cap > 0 && (level[curr] + 1 == level[
92                     edge.to])) {
93                     int bottle_neck = dfs(edge.to, sink,
94                         next, min(flow, cap));
95                     if (bottle_neck > 0) {
96                         edge.flow += bottle_neck;
97                         rev_edge.flow -= bottle_neck;
98                         return bottle_neck;
99                     }
100                 }
101             }
102             return 0;
103         }
104
105         vector<pair<int, int>> mincut(int source, int
106             sink)
107         {
108             vector<pair<int, int>> cut;
109             bfs(source, sink);
110             for (auto & e: edges) {
111                 if (e.flow == e.capacity && level[e.from]
112                     != -1 && level[e.to] == -1 && e.capacity > 0) {
113                     cut.emplace_back(e.from, e.to);
114                 }
115             }
116             return cut;
117         }
118     }

```

```

107     }
108 }
109     return cut;
110 }
111 };
112
113 // Example on how to use
114 void solve()
115 {
116     int n, m;
117     cin >> n >> m;
118     int N = n + m + 2;
119
120     int source = N - 2;
121     int sink = N - 1;
122
123     Dinic flow(N);
124
125     for (int i = 0; i < n; i++) {
126         int q; cin >> q;
127         while(q--) {
128             int b; cin >> b;
129             flow.add_edge(i, n + b - 1, 1);
130         }
131     }
132     for (int i = 0; i < n; i++) {
133         flow.add_edge(source, i, 1);
134     }
135     for (int i = 0; i < m; i++) {
136         flow.add_edge(i + n, sink, 1);
137     }
138
139     cout << m - flow.get_max_flow(source, sink) << endl;
140
141     // Getting participant edges
142     for (auto & edge: flow.edges) {
143         if (edge.capacity == 0) continue; // This
144         // means is a reverse edge
145         if (edge.from == source || edge.to == source) continue;
146         if (edge.from == sink || edge.to == sink) continue;
147         if (edge.flow == 0) continue; // Is not
148         // participant
149         cout << edge.from + 1 << " " << edge.to - n + 1 << endl;
150     }
151 }

```

## 2 Segtree

### 2.1 Standard SegTree

Complexity:  $O(\log(n))$  query and update  
Sum segment tree with point update.

```

1 // TITLE: Standard SegTree
2 // COMPLEXITY:  $O(\log(n))$  query and update
3 // DESCRIPTION: Sum segment tree with point update.
4
5 using type = int;
6
7 type iden = 0;
8 vector<type> seg;
9 int segsize;
10
11 type func(type a, type b)
12 {

```

```

13     return a + b;
14 }
15
16 // query do intervalo [l, r)
17 type query(int l, int r, int no = 0, int lx = 0, int rx = segsize)
18 {
19     // l lx rx r
20     if (r <= lx or rx <= l)
21         return iden;
22     if (l <= lx and rx <= r)
23         return seg[no];
24
25     int mid = lx + (rx - lx) / 2;
26     return func(query(l, r, 2 * no + 1, lx, mid),
27                query(l, r, 2 * no + 2, mid, rx));
28 }
29
30 void update(int dest, type val, int no = 0, int lx = 0, int rx = segsize)
31 {
32     if (dest < lx or dest >= rx)
33         return;
34     if (rx - lx == 1)
35     {
36         seg[no] = val;
37         return;
38     }
39
40     int mid = lx + (rx - lx) / 2;
41     update(dest, val, 2 * no + 1, lx, mid);
42     update(dest, val, 2 * no + 2, mid, rx);
43     seg[no] = func(seg[2 * no + 1], seg[2 * no + 2]);
44 }
45
46 signed main()
47 {
48     ios_base::sync_with_stdio(0);
49     cin.tie(0);
50     cout.tie(0);
51     int n;
52     cin >> n;
53     segsize = n;
54     if (__builtin_popcount(n) != 1)
55     {
56         segsize = 1 + (int)log2(segsize);
57         segsize = 1 << segsize;
58     }
59     seg.assign(2 * segsize - 1, iden);
60
61     rep(i, 0, n)
62     {
63         int x;
64         cin >> x;
65         update(i, x);
66     }
67 }

```

### 2.2 Persistent sum segment tree

Complexity:  $O(\log(n))$  query and update,  $O(k \log(n))$  memory,  
 $n$  = number of elements,  $k$  = number of operations  
Sum segment tree which preserves its history.

```

1 // TITLE: Persistent sum segment tree
2 // COMPLEXITY:  $O(\log(n))$  query and update,  $O(k \log(n))$ 
3 // memory,  $n$  = number of elements,  $k$  = number of
4 // operations
5 // DESCRIPTION: Sum segment tree which preserves its
6 // history.

```

```

5 int segsize;
6
7 struct node {
8     int val;
9     int lx, rx;
10    node *l=0, *r=0;
11
12    node() {}
13    node(int val, int lx, int rx, node *l, node *r) :
14        val(val), lx(lx), rx(rx), l(l), r(r) {}
15 };
16
17 node* build(vi& arr, int lx=0, int rx=segsize) {
18     if (rx - lx == 1) {
19         if (lx < (int)arr.size()) {
20             return new node(arr[lx], lx, rx, 0, 0);
21         }
22     }
23     return new node(0, lx, rx, 0, 0);
24 }
25
26 int mid = (lx+rx)/2;
27 auto nol = build(arr, lx, mid);
28 auto nor = build(arr, mid, rx);
29 return new node(nol->val + nor->val, lx, rx, nol,
30 nor);
31 }
32
33 node* update(int idx, int val, node *no) {
34     if (idx < no->lx or idx >= no->rx) return no;
35     if (no->rx - no->lx == 1) {
36         return new node(val+no->val, no->lx, no->rx,
37 no->l, no->r);
38     }
39
40     auto nol = update(idx, val, no->l);
41     auto nor = update(idx, val, no->r);
42     return new node(nol->val + nor->val, no->lx, no->rx,
43 nol, nor);
44 }
45
46 int query(int l, int r, node *no) {
47     if (r <= no->lx or no->rx <= l) return 0;
48     if (l <= no->lx and no->rx <= r) return no->val;
49
50     return query(l, r, no->l) + query(l, r, no->r);
51 }

```

## 2.3 Set and update lazy seg

Complexity:  $O(\log(n))$  query and update  
Sum segtree with set and update

```

1 // TITLE: Set and update lazy seg
2 // COMPLEXITY:  $O(\log(n))$  query and update
3 // DESCRIPTION: Sum segtree with set and update
4
5 vector<int> lazy, opvec;
6 vector<int> seg;
7
8 constexpr int SET = 30;
9 constexpr int ADD = 31;
10
11 int segsize;
12
13 void propagate(int no, int lx, int rx) {
14     if (lazy[no] == -1) return;
15
16     if (rx-lx == 1) {
17         if (opvec[no] == SET) seg[no] = lazy[no];
18         else seg[no] += lazy[no];
19     }

```

```

20     lazy[no] = -1;
21     opvec[no] = -1;
22     return;
23 }
24
25 if (opvec[no] == SET) {
26     seg[no] = (rx-lx) * lazy[no];
27     lazy[2*no+1] = lazy[no];
28     lazy[2*no+2] = lazy[no];
29
30     opvec[2*no+1] = SET;
31     opvec[2*no+2] = SET;
32
33     lazy[no] = -1;
34     opvec[no] = -1;
35     return;
36 }
37
38 seg[no] += (rx-lx) * lazy[no];
39 if (lazy[2*no+1] == -1) {
40     lazy[2*no+1] = 0;
41     opvec[2*no+1] = ADD;
42 }
43 if (lazy[2*no+2] == -1) {
44     lazy[2*no+2] = 0;
45     opvec[2*no+2] = ADD;
46 }
47 lazy[2*no+1] += lazy[no];
48 lazy[2*no+2] += lazy[no];
49
50 lazy[no] = -1;
51 opvec[no] = -1;
52 }
53
54 void update(int l, int r, int val, int op, int no=0,
55 int lx=0, int rx=segsize) {
56     propagate(no, lx, rx);
57     if (r <= lx or l >= rx) return;
58     if (lx >= l and rx <= r) {
59         lazy[no] = val;
60         opvec[no] = op;
61         propagate(no, lx, rx);
62         return;
63     }
64
65     int mid = (rx+lx)/2;
66     update(l, r, val, op, 2*no+1, lx, mid);
67     update(l, r, val, op, 2*no+2, mid, rx);
68     seg[no] = seg[2*no+1] + seg[2*no+2];
69 }
70
71 int query(int l, int r, int no=0, int lx=0, int rx=
72 segsize) {
73     propagate(no, lx, rx);
74     if (r <= lx or l >= rx) return 0;
75     if (lx >= l and rx <= r) return seg[no];
76
77     int mid = (rx+lx)/2;
78     return
79         query(l, r, 2*no+1, lx, mid) +
80         query(l, r, 2*no+2, mid, rx);
81 }

```

## 2.4 Lazy SegTree

Complexity:  $O(\log(n))$  query and update  
Sum segment tree with range sum update.

```

1 // TITLE: Lazy SegTree
2 // COMPLEXITY:  $O(\log(n))$  query and update
3 // DESCRIPTION: Sum segment tree with range sum
  update.

```



```

4 vector<int> seg, lazy;
5 int segsize;
6
7 // change 0s to -1s if update is
8 // set instead of add. also,
9 // remove the +=s
10 void prop(int no, int lx, int rx) {
11     if (lazy[no] == 0) return;
12
13     seg[no]+=(rx-lx)*lazy[no];
14     if (rx-lx>1) {
15         lazy[2*no+1] += lazy[no];
16         lazy[2*no+2] += lazy[no];
17     }
18
19     lazy[no]=0;
20 }
21
22 void update(int l, int r, int val, int no=0, int lx=0,
23             int rx=segsize) {
24     // l r lx rx
25     prop(no, lx, rx);
26     if (r <= lx or rx <= l) return;
27     if (l <= lx and rx <= r) {
28         lazy[no]=val;
29         prop(no, lx, rx);
30         return;
31     }
32
33     int mid=lx+(rx-lx)/2;
34     update(l, r, val, 2*no+1, lx, mid);
35     update(l, r, val, 2*no+2, mid, rx);
36     seg[no] =seg[2*no+1]+seg[2*no+2];
37 }
38
39 int query(int l, int r, int no=0, int lx=0, int rx=
40           segsize) {
41     prop(no, lx, rx);
42     if (r <= lx or rx <= l) return 0;
43     if (l <= lx and rx <= r) return seg[no];
44
45     int mid=lx+(rx-lx)/2;
46     return query(l, r, 2*no+1, lx, mid)+
47            query(l, r, 2*no+2, mid, rx);
48 }
49
50 signed main() {
51     ios_base::sync_with_stdio(0);cin.tie(0);cout.tie
52     (0);
53
54     int n;cin>>n;
55     segsize=n;
56     if (__builtin_popcount(n) != 1) {
57         segsize=1+(int)log2(segsize);
58         segsize= 1<<segsize;
59     }
60
61     seg.assign(2*segsize-1, 0);
62     // use -1 instead of 0 if
63     // update is set instead of add
64     lazy.assign(2*segsize-1, 0);
65 }

```

## 3 Set

### 3.1 Set

Complexity: Insertion Log(n)

Keeps elements sorted, remove duplicates, upper\_bound, lower\_bound, find, count

```

1 // TITLE: Set
2 // COMPLEXITY: Insertion Log(n)
3 // Description: Keeps elements sorted, remove
4 // duplicates, upper_bound, lower_bound, find, count
5
6 int main() {
7     set<int> set1;
8
9     set1.insert(1); // 0(log(n))
10    set1.erase(1); // 0(log(n))
11
12    set1.upper_bound(1); // 0(log(n))
13    set1.lower_bound(1); // 0(log(n))
14    set1.find(1); // 0(log(n))
15    set1.count(1); // 0(log(n))
16
17    set1.size(); // 0(1)
18    set1.empty(); // 0(1)
19
20    set1.clear() // 0(1)
21    return 0;
22 }

```

### 3.2 Multiset

Complexity: O(log(n))

Same as set but you can have multiple elements with same values

```

1 // TITLE: Multiset
2 // COMPLEXITY: O(log(n))
3 // DESCRIPTION: Same as set but you can have multiple
4 // elements with same values
5
6 int main() {
7     multiset<int> set1;
8 }

```

### 3.3 Ordered Set

Complexity: log n

Worst set with adtional operations

```

1 // TITLE: Ordered Set
2 // COMPLEXITY: log n
3 // DESCRIPTION: Worst set with adtional operations
4
5
6 #include <bits/extc++.h>
7 using namespace __gnu_pbds; // or pb_ds;
8 template<typename T, typename B = null_type>
9 using ordered_set = tree<T, B, less<T>, rb_tree_tag,
10 tree_order_statistics_node_update>;
11
12 int32_t main() {
13     ordered_set<int> oset;
14
15     oset.insert(5);
16     oset.insert(1);
17     oset.insert(2);
18     // o_set = {1, 2, 5}
19     5 == *(oset.find_by_order(2)); // Like an array
20     index
21     2 == oset.order_of_key(4); // How many elements
22     are strictly less than 4

```

## 4 Misc

### 4.1 Template

Complexity:  $O(1)$

Standard template for competitions

```
1 // TITLE: Template
2 // COMPLEXITY:  $O(1)$ 
3 // DESCRIPTION: Standard template for competitions
4
5 #include <bits/stdc++.h>
6
7 #define int long long
8 #define endl '\n'
9 #define pb push_back
10 #define eb emplace_back
11 #define all(x) (x).begin(), (x).end()
12 #define rep(i, a, b) for(int i=(int)(a); i < (int)(b); i++)
13 #define debug(var) cout << #var << ": " << var << endl
14 #define pii pair<int, int>
15 #define vi vector<int>
16
17 int MAX = 2e5;
18 int MOD=1e9+7;
19 int oo=0x3f3f3f3f3f3f3f3f;
20
21 using namespace std;
22
23 void solve()
24 {
25
26 }
27
28 signed main()
29 {
30     ios_base::sync_with_stdio(0); cin.tie(0); cout.tie(0);
31     int t=1;
32     // cin>>t;
33     while(t--) solve();
34 }
```

## 5 String

### 5.1 String hash

Complexity:  $O(n)$  preprocessing,  $O(1)$  query

Computes the hash of arbitrary substrings of a given string s.

```
1 // TITLE: String hash
2 // COMPLEXITY:  $O(n)$  preprocessing,  $O(1)$  query
3 // DESCRIPTION: Computes the hash of arbitrary
  substrings of a given string s.
4
5 struct hashes
6 {
7     string s;
8     int m1, m2, n, p;
9     vector<int> p1, p2, sum1, sum2;
10
11     hashes(string s) : s(s), n(s.size()), p1(n + 1),
12     p2(n + 1), sum1(n + 1), sum2(n + 1)
13     {
14         srand(time(0));
15         p = 31;
16         m1 = rand() / 10 + 1e9; // 1000253887;
17         m2 = rand() / 10 + 1e9; // 1000546873;
```

```
17
18     p1[0] = p2[0] = 1;
19     rep(i, 1, n + 1)
20     {
21         p1[i] = (p * p1[i - 1]) % m1;
22         p2[i] = (p * p2[i - 1]) % m2;
23     }
24
25     sum1[0] = sum2[0] = 0;
26     rep(i, 1, n + 1)
27     {
28         sum1[i] = (sum1[i - 1] * p) % m1 + s[i -
29     1];
30         sum2[i] = (sum2[i - 1] * p) % m2 + s[i -
31     1];
32         sum1[i] %= m1;
33         sum2[i] %= m2;
34     }
35
36     // hash do intervalo [l, r)
37     int gethash(int l, int r)
38     {
39         int c1 = m1 - (sum1[l] * p1[r - 1]) % m1;
40         int c2 = m2 - (sum2[l] * p2[r - 1]) % m2;
41         int h1 = (sum1[r] + c1) % m1;
42         int h2 = (sum2[r] + c2) % m2;
43         return (h1 << 30) ^ h2;
44     }
45 };
```

### 5.2 Suffix Array

Complexity:  $O(n \log(n))$ , contains big constant (around 25).

Computes a sorted array of the suffixes of a string.

```
1 // TITLE: Suffix Array
2 // COMPLEXITY:  $O(n \log(n))$ , contains big constant (
  around 25).
3 // DESCRIPTION: Computes a sorted array of the
  suffixes of a string.
4
5 void countingsort(vi& p, vi& c) {
6     int n=p.size();
7     vi count(n,0);
8     rep(i,0,n) count[c[i]]++;
9
10     vi psum(n); psum[0]=0;
11     rep(i,1,n) psum[i]=psum[i-1]+count[i-1];
12
13     vi ans(n);
14     rep(i,0,n)
15         ans[psum[c[p[i]]]++]=p[i];
16
17     p = ans;
18 }
19
20 vi sfa(string s) {
21     s += "$";
22
23     int n=s.size();
24     vi p(n);
25     vi c(n);
26     {
27         vector<pair<char, int>> a(n);
28         rep(i,0,n) a[i]={s[i],i};
29         sort(all(a));
30
31         rep(i,0,n) p[i]=a[i].second;
32
33         c[p[0]]=0;
34         rep(i,1,n) {
```

```

35         if(s[p[i]] == s[p[i-1]]) {
36             c[p[i]]=c[p[i-1]];
37         }
38         else c[p[i]]=c[p[i-1]]+1;
39     }
40 }
41
42 for(int k=0; (1<<k) < n; k++) {
43     rep(i, 0, n)
44         p[i] = (p[i] - (1<<k) + n) % n;
45
46     countingsort(p,c);
47
48     vi nc(n);
49     nc[p[0]]=0;
50     rep(i,1,n) {
51         pii prev = {c[p[i-1]], c[(p[i-1]+(1<<k))%n]};
52     };
53     pii cur = {c[p[i]], c[(p[i]+(1<<k))%n]};
54
55     if (prev == cur)
56         nc[p[i]]=nc[p[i-1]];
57     else nc[p[i]]=nc[p[i-1]]+1;
58 }
59 c=nc;
60 return p;
61 }

```

## 5.3 Z function

Complexity: Z function complexity  
z function

```

1 // TITLE: Z function
2 // COMPLEXITY: Z function complexity
3 // DESCRIPTION: z function
4
5 void z_function(string& s)
6 {
7     return;
8 }

```

# 6 Algorithms

## 6.1 Sparse table

Complexity:  $O(n \log(n))$  preprocessing,  $O(1)$  query  
Computes the minimum of a half open interval.

```

1 // TITLE: Sparse table
2 // COMPLEXITY:  $O(n \log(n))$  preprocessing,  $O(1)$  query
3 // DESCRIPTION: Computes the minimum of a half open
  interval.
4
5 struct sptable {
6     vector<vi> table;
7
8     int ilog(int x) {
9         return (__builtin_clzll(1ll) -
10             __builtin_clzll(x));
11     }
12
13     sptable(vi& vals) {
14         int n = vals.size();
15         int ln= ilog(n)+1;
16         table.assign(ln, vi(n));

```

```

17         rep(i,0,n) table[0][i]=vals[i];
18
19         rep(k, 1, ln) {
20             rep(i,0,n) {
21                 table[k][i] = min(table[k-1][i],
22                     table[k-1][min(i + (1<<(k-1)), n-1)])
23             };
24         }
25     }
26
27     // returns minimum of vals in range [a, b)
28     int getmin(int a, int b) {
29         int k = ilog(b-a);
30         return min(table[k][a], table[k][b-(1<<k)]);
31     }

```

## 7 Parser

### 7.1 Parsing Functions

Complexity:

```

1 // TITLE: Parsing Functions
2
3 vector<string> split_string(const string & s, const
4     string & sep = " ") {
5     int w = sep.size();
6     vector<string> ans;
7     string curr;
8
9     auto add = [&](string a) {
10         if (a.size() > 0) {
11             ans.push_back(a);
12         }
13     };
14
15     for (int i = 0; i + w < s.size(); i++) {
16         if (s.substr(i, w) == sep) {
17             i += w-1;
18             add(curr);
19             curr.clear();
20             continue;
21         }
22         curr.push_back(s[i]);
23     }
24     add(curr);
25     return ans;
26 }
27
28 vector<int> parse_vector_int(string & s)
29 {
30     vector<int> nums;
31     for (string x: split_string(s)) {
32         nums.push_back(stoi(x));
33     }
34     return nums;
35 }
36
37 vector<float> parse_vector_float(string & s)
38 {
39     vector<float> nums;
40     for (string x: split_string(s)) {
41         nums.push_back(stof(x));
42     }
43     return nums;
44 }

```

```
45 void solve()
46 {
47     cin.ignore();
48     string s;
49     getline(cin, s);
50
```

```
51     auto nums = parse_vector_float(s);
52     for (auto x: nums) {
53         cout << x << endl;
54     }
55 }
```