



Notebook - Maratona de Programação

Cabo HDMI, VGA, USB

Contents

1	Segtree	2
1.1	Standard SegTree	2
1.2	Lazy SegTree	2
2	Graph	3
2.1	Dinic	3
2.2	Dinic Min cost	4
2.3	Dijkstra	5
2.4	Dominator tree	5
2.5	Bellman Ford	6
2.6	Dfs tree	6
2.7	Topological Sort	6
2.8	Kth Ancestor	7
3	Parser	7
3.1	Parsing Functions	7
4	String	8
4.1	Z function	8
4.2	Suffix Array	8
4.3	String hash	8
5	Algorithms	9
5.1	Sparse table	9
6	Set	9
6.1	Ordered Set	9
6.2	Set	9
6.3	Multiset	9

1 Segtree

1.1 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     loop(i, 0, n)
62     {
63         int x;
64         cin >> x;
```

```
65         update(i, x);
66     }
67 }
```

1.2 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,
    int rx = segsize) {
23     // l r lx rx
24     prop(no, lx, rx);
25     if (r <= lx or rx <= l) return;
26     if (l <= lx and rx <= r) {
27         lazy[no] = val;
28         prop(no, lx, rx);
29         return;
30     }
31
32     int mid = lx + (rx - lx) / 2;
33     update(l, r, val, 2 * no + 1, lx, mid);
34     update(l, r, val, 2 * no + 2, mid, rx);
35     seg[no] = seg[2 * no + 1] + seg[2 * no + 2];
36 }
37
38 int query(int l, int r, int no = 0, int lx = 0, int rx =
    segsize) {
39     prop(no, lx, rx);
40     if (r <= lx or rx <= l) return 0;
41     if (l <= lx and rx <= r) return seg[no];
42
43     int mid = lx + (rx - lx) / 2;
44     return query(l, r, 2 * no + 1, lx, mid) +
45         query(l, r, 2 * no + 2, mid, rx);
46 }
47
48 signed main() {
49     ios_base::sync_with_stdio(0); cin.tie(0); cout.tie
    (0);
50
51     int n; cin >> n;
52     segsize = n;
53     if (__builtin_popcount(n) != 1) {
54         segsize = 1 + (int)log2(segsize);
55         segsize = 1 << segsize;
56     }
57
58     seg.assign(2 * segsize - 1, 0);
59     // use -1 instead of 0 if
```

```

60 // update is set instead of add
61 lazy.assign(2*segsz-1, 0);
62 }

```

2 Graph

2.1 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         bool bfs(int source, int sink)
55         {
56             level.assign(size, -1);

```

```

57     queue<int> q;
58     q.push(source);
59     level[source] = 0;
60
61     while(!q.empty()) {
62         int a = q.front();
63         q.pop();
64
65         for (int & b: graph[a]) {
66             auto edge = edges[b];
67             int cap = edge.capacity - edge.flow;
68             if (cap > 0 && level[edge.to] == -1)
69             {
70                 level[edge.to] = level[a] + 1;
71                 q.push(edge.to);
72             }
73         }
74         return level[sink] != -1;
75     }
76
77     int dfs(int curr, int sink, vector<int> & next,
78             int flow)
79     {
80         if (curr == sink) return flow;
81         int num_edges = graph[curr].size();
82
83         for (; next[curr] < num_edges; next[curr]++)
84         {
85             int b = graph[curr][next[curr]];
86             auto & edge = edges[b];
87             auto & rev_edge = edges[b^1];
88
89             int cap = edge.capacity - edge.flow;
90             if (cap > 0 && (level[curr] + 1 == level[
91                 edge.to])) {
92                 int bottle_neck = dfs(edge.to, sink,
93                     next, min(flow, cap));
94                 if (bottle_neck > 0) {
95                     edge.flow += bottle_neck;
96                     rev_edge.flow -= bottle_neck;
97                     return bottle_neck;
98                 }
99             }
100             return 0;
101         }
102     }
103
104     vector<pair<int, int>> mincut(int source, int
105     sink)
106     {
107         vector<pair<int, int>> cut;
108         bfs(source, sink);
109         for (auto & e: edges) {
110             if (e.flow == e.capacity && level[e.from]
111                 != -1 && level[e.to] == -1 && e.capacity > 0) {
112                 cut.emplace_back(e.from, e.to);
113             }
114         }
115         return cut;
116     }
117 };
118
119 // Example on how to use
120 void solve()
121 {
122     int n, m;
123     cin >> n >> m;
124     int N = n + m + 2;
125
126     int source = N - 2;
127     int sink = N - 1;

```

```

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
means is a reverse edge
144     if (edge.from == source || edge.to == source)
continue;
145     if (edge.from == sink || edge.to == sink)
continue;
146     if (edge.flow == 0) continue; // Is not
participant
147
148     cout << edge.from + 1 << " " << edge.to - n +
1 << endl;
149 }
150 }

```

2.2 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_,
int cost_)
13        : from(from_), to(to_), flow(flow_), capacity
(capacity_), cost(cost_)
14    {}
15 };
16
17 struct Dinic
18 {
19     vector<vector<int>> graph;
20     vector<Edge> edges;
21     vector<int> dist;
22     vector<bool> inqueue;
23     int size;
24     int cost = 0;
25
26     Dinic(int n)
27     {
28         graph.resize(n);
29         dist.resize(n);

```

```

30         inqueue.resize(n);
31         size = n;
32         edges.clear();
33     }
34
35     void add_edge(int from, int to, int capacity, int
cost)
36     {
37         edges.emplace_back(from, to, 0, capacity,
cost);
38         graph[from].push_back(edges.size() - 1);
39
40         edges.emplace_back(to, from, 0, 0, -cost);
41         graph[to].push_back(edges.size() - 1);
42     }
43
44     int get_max_flow(int source, int sink)
45     {
46         int max_flow = 0;
47         vector<int> next(size);
48         while(spfa(source, sink)) {
49             next.assign(size, 0);
50             for (int f = dfs(source, sink, next, oo);
f != 0; f = dfs(source, sink, next, oo)) {
51                 max_flow += f;
52             }
53         }
54         return max_flow;
55     }
56
57     bool spfa(int source, int sink)
58     {
59         dist.assign(size, oo);
60         inqueue.assign(size, false);
61         queue<int> q;
62         q.push(source);
63         dist[source] = 0;
64         inqueue[source] = true;
65
66         while(!q.empty()) {
67             int a = q.front();
68             q.pop();
69             inqueue[a] = false;
70
71             for (int & b: graph[a]) {
72                 auto edge = edges[b];
73                 int cap = edge.capacity - edge.flow;
74                 if (cap > 0 && dist[edge.to] > dist[
edge.from] + edge.cost) {
75                     dist[edge.to] = dist[edge.from] +
edge.cost;
76                     if (not inqueue[edge.to]) {
77                         q.push(edge.to);
78                         inqueue[edge.to] = true;
79                     }
80                 }
81             }
82         }
83         return dist[sink] != oo;
84     }
85
86     int dfs(int curr, int sink, vector<int> & next,
int flow)
87     {
88         if (curr == sink) return flow;
89         int num_edges = graph[curr].size();
90
91         for (; next[curr] < num_edges; next[curr]++)
92         {
93             int b = graph[curr][next[curr]];
94             auto & edge = edges[b];
95             auto & rev_edge = edges[b^1];

```

```

96         int cap = edge.capacity - edge.flow;
97         if (cap > 0 && (dist[edge.from] + edge.
cost == dist[edge.to])) {
98             int bottle_neck = dfs(edge.to, sink,
next, min(flow, cap));
99             if (bottle_neck > 0) {
100                 edge.flow += bottle_neck;
101                 rev_edge.flow -= bottle_neck;
102                 cost += edge.cost * bottle_neck;
103                 return bottle_neck;
104             }
105         }
106     }
107     return 0;
108 }
109
110 vector<pair<int, int>> mincut(int source, int
sink)
111 {
112     vector<pair<int, int>> cut;
113     spfa(source, sink);
114     for (auto & e: edges) {
115         if (e.flow == e.capacity && dist[e.from]
!= oo && level[e.to] == oo && e.capacity > 0) {
116             cut.emplace_back(e.from, e.to);
117         }
118     }
119     return cut;
120 }
121 };
122
123 // Example on how to use
124 void solve()
125 {
126
127     int N = 10;
128
129     int source = 8;
130     int sink = 9;
131
132     Dinic flow(N);
133     flow.add_edge(8, 0, 4, 0);
134     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 }

```

2.3 Dkistra

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

```

1 // TITLE: Dkistra
2 // COMPLEXITY:  $O(E + V \log(v))$ 
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 }

```

2.4 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 }

```

2.5 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. The node can have negative cycle and belman-ford will detected

```

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. The node can have
5 // negative cycle and belman-ford will detected
6 // a and b vertices, c cost
7 // [{a, b, c}, {a, b, c}]
8 vector<tuple<int, int, int>> edges;
9 int N;
10
11 void bellman_ford(int x){
12     for (int i = 0; i < N; i++){
13         dist[i] = oo;
14     }
15     dist[x] = 0;

```

```

16     for (int i = 0; i < N - 1; i++){
17         for (auto [a, b, c]: edges){
18             if (dist[a] == oo) continue;
19             dist[b] = min(dist[b], dist[a] + w);
20         }
21     }
22 }
23 // return true if has cycle
24 bool check_negative_cycle(int x){
25     for (int i = 0; i < N; i++){
26         dist[i] = oo;
27     }
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 '''

```

2.6 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 }

```

2.7 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
  topologica, se a quantidade de nos retornada nao
  for igual a quantidade de nos e impossivel
4
5 typedef vector<vector<int>> Adj_List;
6 typedef vector<int> Indegree_List; // How many nodes
  depend on him
7 typedef vector<int> Order_List;    // The order in
  which the nodes appears
8
9 Order_List kahn(Adj_List adj, Indegree_List indegree)
10 {
11     queue<int> q;
12     // priority_queue<int> q; // If you want in
  lexicografic order
13     for (int i = 0; i < indegree.size(); i++) {
14         if (indegree[i] == 0)
15             q.push(i);
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
36     Order_List = kahn(adj, indegree);
37     if (Order_List.size() != N) {
38         cout << "IMPOSSIBLE" << endl;
39     }
40     return 0;
41 }

```

2.8 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,
  int k)
7 {
8     for (int j = 0; j < LOG_N; j++) {
9         if (k & ((int)1 << j)) {
10             v = up[v][j];
11         }
12     }
13     return v;
14 }
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 }

```

3 Parser

3.1 Parsing Functions

Complexity:

```

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

```

4 String

4.1 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 }

```

4.2 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 (
3 // DESCRIPTION: Computes a sorted array of the
4 // suffixes of a string.
5
6 void countingsort(vi& p, vi& c) {
7     int n=p.size();
8     vi count(n,0);
9     loop(i,0,n) count[c[i]]++;
10
11     vi psum(n); psum[0]=0;
12     loop(i,1,n) psum[i]=psum[i-1]+count[i-1];
13
14     vi ans(n);
15     loop(i,0,n)
16         ans[psum[c[p[i]]]++]=p[i];
17
18     p = ans;
19 }
20
21 vi sfa(string s) {
22     s += "$";
23
24     int n=s.size();
25     vi p(n);
26     vi c(n);
27     {
28         vector<pair<char, int>> a(n);
29         loop(i,0,n) a[i]={s[i],i};
30         sort(all(a));
31
32         loop(i,0,n) p[i]=a[i].second;
33
34         c[p[0]]=0;
35         loop(i,1,n) {
36             if(s[p[i]] == s[p[i-1]]) {
37                 c[p[i]]=c[p[i-1]];
38             }
39             else c[p[i]]=c[p[i-1]]+1;
40         }
41
42         for(int k=0; (1<<k) < n; k++) {

```

```

43         loop(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         loop(i,1,n) {
51             pii prev = {c[p[i-1]], c[(p[i-1]+(1<<k))%
52             n]};
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     }
61     return p;
62 }

```

4.3 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
4 // substrings of a given string s.
5
6 struct hashes
7 {
8     string s;
9     int m1, m2, n, p;
10     vector<int> p1, p2, sum1, sum2;
11
12     hashes(string s) : s(s), n(s.size()), p1(n + 1),
13     p2(n + 1), sum1(n + 1), sum2(n + 1)
14     {
15         srand(time(0));
16         p = 31;
17         m1 = rand() / 10 + 1e9; // 1000253887;
18         m2 = rand() / 10 + 1e9; // 1000546873;
19
20         p1[0] = p2[0] = 1;
21         loop(i, 1, n + 1)
22         {
23             p1[i] = (p * p1[i - 1]) % m1;
24             p2[i] = (p * p2[i - 1]) % m2;
25         }
26
27         sum1[0] = sum2[0] = 0;
28         loop(i, 1, n + 1)
29         {
30             sum1[i] = (sum1[i - 1] * p) % m1 + s[i -
31             1];
32             sum2[i] = (sum2[i - 1] * p) % m2 + s[i -
33             1];
34             sum1[i] %= m1;
35             sum2[i] %= m2;
36         }
37     }
38
39     // hash do intervalo [l, r)
40     int gethash(int l, int r)
41     {
42         int c1 = m1 - (sum1[r] * p1[r - l]) % m1;
43         int c2 = m2 - (sum2[r] * p2[r - l]) % m2;
44         int h1 = (sum1[l] + c1) % m1;
45         int h2 = (sum2[l] + c2) % m2;

```



```

42     return (h1 << 30) ^ h2;
43 }
44 };

```

5 Algorithms

5.1 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
18         loop(i,0,n) table[0][i]=vals[i];
19
20         loop(k, 1, ln) {
21             loop(i,0,n) {
22                 table[k][i] = min(table[k-1][i],
23                 table[k-1][min(i + (1<<(k-1)), n-1)]);
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         }
32 };

```

6 Set

6.1 Ordered Set

Complexity: $\log n$

Worst set with additional operations

```

1 // TITLE: Ordered Set
2 // COMPLEXITY:  $\log n$ 
3 // DESCRIPTION: Worst set with additional 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,
  tree_order_statistics_node_update>;
10
11 int32_t main() {
12     ordered_set<int> oset;
13
14     oset.insert(5);
15     oset.insert(1);
16     oset.insert(2);
17     // o_set = {1, 2, 5}
18     5 == *(oset.find_by_order(2)); // Like an array
  index
19     2 == oset.order_of_key(4); // How many elements
  are strictly less than 4
20 }

```

6.2 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
  duplicates, upper_bound, lower_bound, find, count
4
5 int main() {
6     set<int> set1;
7
8     set1.insert(1); //  $O(\log(n))$ 
9     set1.erase(1); //  $O(\log(n))$ 
10
11     set1.upper_bound(1); //  $O(\log(n))$ 
12     set1.lower_bound(1); //  $O(\log(n))$ 
13     set1.find(1); //  $O(\log(n))$ 
14     set1.count(1); //  $O(\log(n))$ 
15
16     set1.size(); //  $O(1)$ 
17     set1.empty(); //  $O(1)$ 
18
19     set1.clear() //  $O(1)$ 
20     return 0;
21 }

```

6.3 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
  elements with same values
4
5 int main() {
6     multiset<int> set1;
7 }

```