



Notebook - Maratona de Programação

Cabo HDMI, VGA, USB

Contents

1	String	2
1.1	String hash	2
1.2	Z function	2
2	Segtree	2
2.1	Standard SegTree	2
2.2	3
3	Set	3
3.1	Ordered Set	3
3.2	Multiset	3
3.3	Set	3
4	Graph	3
4.1	Topological Sort	3
4.2	Dinic	4
4.3	Bellman Ford	5

1 String

1.1 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;
18
19         p1[0] = p2[0] = 1;
20         loop(i, 1, n + 1)
21         {
22             p1[i] = (p * p1[i - 1]) % m1;
23             p2[i] = (p * p2[i - 1]) % m2;
24         }
25
26         sum1[0] = sum2[0] = 0;
27         loop(i, 1, n + 1)
28         {
29             sum1[i] = (sum1[i - 1] * p) % m1 + s[i -
30             1];
31             sum2[i] = (sum2[i - 1] * p) % m2 + s[i -
32             1];
33             sum1[i] %= m1;
34             sum2[i] %= m2;
35         }
36
37         // hash do intervalo [l, r)
38         int gethash(int l, int r)
39         {
40             int c1 = m1 - (sum1[l] * p1[r - 1]) % m1;
41             int c2 = m2 - (sum2[l] * p2[r - 1]) % m2;
42             int h1 = (sum1[r] + c1) % m1;
43             int h2 = (sum2[r] + c2) % m2;
44             return (h1 << 30) ^ h2;
45         }
46     };
47 }
```

1.2 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 }
```

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
18           rx = segsize)
19 {
20     // l lx rx r
21     if (r <= lx or rx <= l)
22         return iden;
23     if (l <= lx and rx <= r)
24         return seg[no];
25
26     int mid = lx + (rx - lx) / 2;
27     return func(query(l, r, 2 * no + 1, lx, mid),
28               query(l, r, 2 * no + 2, mid, rx));
29 }
30
31 void update(int dest, type val, int no = 0, int lx =
32           0, int rx = segsize)
33 {
34     if (dest < lx or dest >= rx)
35         return;
36     if (rx - lx == 1)
37     {
38         seg[no] = val;
39         return;
40     }
41
42     int mid = lx + (rx - lx) / 2;
43     update(dest, val, 2 * no + 1, lx, mid);
44     update(dest, val, 2 * no + 2, mid, rx);
45     seg[no] = func(seg[2 * no + 1], seg[2 * no + 2]);
46 }
47
48 signed main()
49 {
50     ios_base::sync_with_stdio(0);
51     cin.tie(0);
52     cout.tie(0);
53
54     int n;
55     cin >> n;
56     segsize = n;
57     if (__builtin_popcount(n) != 1)
58     {
59         segsize = 1 + (int)log2(segsize);
60         segsize = 1 << segsize;
61     }
62     seg.assign(2 * segsize - 1, iden);
63
64     loop(i, 0, n)
65     {
66         int x;
67         cin >> x;
68     }
69 }
```

```

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

```

2.2

Complexity:

```

1  using type = int;
2
3  type iden = 0;
4  vector<type> seg;
5  int segsize;
6
7  type func(type a, type b)
8  {
9      return a + b;
10 }
11
12 // query do intervalo [l, r)
13 type query(int l, int r, int no = 0, int lx = 0, int
14            rx = segsize)
15 {
16     // l lx rx r
17     if (r <= lx or rx <= l)
18         return iden;
19     if (l <= lx and rx <= r)
20         return seg[no];
21
22     int mid = lx + (rx - lx) / 2;
23     return func(query(l, r, 2 * no + 1, lx, mid),
24                query(l, r, 2 * no + 2, mid, rx));
25 }
26
27 void update(int dest, type val, int no = 0, int lx =
28            0, int rx = segsize)
29 {
30     if (dest < lx or dest >= rx)
31         return;
32     if (rx - lx == 1)
33     {
34         seg[no] = val;
35         return;
36     }
37
38     int mid = lx + (rx - lx) / 2;
39     update(dest, val, 2 * no + 1, lx, mid);
40     update(dest, val, 2 * no + 2, mid, rx);
41     seg[no] = func(seg[2 * no + 1], seg[2 * no + 2]);
42 }

```

3 Set

3.1 Ordered Set

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

```

1  // TITLE: Ordered Set
2  // COMPLEXITY:  $O(\log(n))$ 
3  // DESCRIPTION: Set but you can look witch elements is
4                  in position (k)
5
6  #include <ext/pb_ds/assoc_container.hpp>
7  #include <ext/pb_ds/tree_policy.hpp>
8  using namespace __gnu_pbds;

```

```

9  #define ordered_set tree<int, null_type, less<int>,
10     rb_tree_tag, tree_order_statistics_node_update>
11
12 int32_t main() {
13     ordered_set o_set;
14
15     o_set.insert(5);
16     o_set.insert(1);
17     o_set.insert(2);
18     // o_set = {1, 2, 5}
19     5 == *(o_set.find_by_order(2));
20     2 == o_set.order_of_key(4); // {1, 2}
21 }

```

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 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);           //  $O(\log(n))$ 
10     set1.erase(1);           //  $O(\log(n))$ 
11
12     set1.upper_bound(1);      //  $O(\log(n))$ 
13     set1.lower_bound(1);      //  $O(\log(n))$ 
14     set1.find(1);             //  $O(\log(n))$ 
15     set1.count(1);            //  $O(\log(n))$ 
16
17     set1.size();              //  $O(1)$ 
18     set1.empty();             //  $O(1)$ 
19
20     set1.clear()              //  $O(1)$ 
21     return 0;
22 }

```

4 Graph

4.1 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 }

```

4.2 Dinic

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

Dinic is a strongly polynomial maximum flow algorithm, doesn't depend on capacity values good for matching

```

1 // TITLE: Dinic
2 // COMPLEXITY:  $O(V \cdot V \cdot E)$ , Bipartite is  $O(\sqrt{V} \cdot E)$ 
3 // DESCRIPTION: Dinic is a strongly polynomial
  maximum flow algorithm, doesn't depend on capacity
  values good for matching
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_
  (capacity_)
14    {}

```

```

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]];

```

```

84         auto & edge = edges[b];
85         auto & rev_edge = edges[b^1];
86
87         int cap = edge.capacity - edge.flow;
88         if (cap > 0 && (level[curr] + 1 == level[
edge.to])) {
89             int bottle_neck = dfs(edge.to, sink,
next, min(flow, cap));
90             if (bottle_neck > 0) {
91                 edge.flow += bottle_neck;
92                 rev_edge.flow -= bottle_neck;
93                 return bottle_neck;
94             }
95         }
96     }
97     return 0;
98 }
99 };
100
101 // Example on how to use
102 void solve()
103 {
104     int n, m;
105     cin >> n >> m;
106     int N = n + m + 2;
107
108     int source = N - 2;
109     int sink = N - 1;
110
111     Dinic flow(N);
112
113     for (int i = 0; i < n; i++) {
114         int q; cin >> q;
115         while(q--) {
116             int b; cin >> b;
117             flow.add_edge(i, n + b - 1, 1);
118         }
119     }
120     for (int i = 0; i < n; i++) {
121         flow.add_edge(source, i, 1);
122     }
123     for (int i = 0; i < m; i++) {
124         flow.add_edge(i + n, sink, 1);
125     }
126
127     cout << m - flow.get_max_flow(source, sink) <<
endl;
128
129     // Getting participant edges
130     for (auto & edge: flow.edges) {
131         if (edge.capacity == 0) continue; // This
means is a reverse edge
132         if (edge.from == source || edge.to == source)
continue;
133         if (edge.from == sink || edge.to == sink)
continue;
134         if (edge.flow == 0) continue; // Is not
participant
135
136         cout << edge.from + 1 << " " << edge.to - n +
1 << endl;

```

```

137     }
138 }

```

4.3 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
node to all nodes of the graph. The node can have
negative cycle and belman-ford will detected
4
5 // a and b vertices, c cost
6 // [{a, b, c}, {a, b, c}]
7 vector<tuple<int, int, int>> edges;
8 int N;
9
10 void bellman_ford(int x){
11     for (int i = 0; i < N; i++){
12         dist[i] = oo;
13     }
14     dist[x] = 0;
15
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 '''

```