

Competitive Programming Algorithms and Topics

Ubiratan Neto

1 Data Structures

- 1.1 Segment Tree using Pointers
- 1.2 Range Update Segment Tree
- 1.3 Range Update Binary Indexed Tree

2 Uncategorized

- 2.1 Longest Increasing Subsequence

3 Geometry

- 3.1 Closest Pair of Points

4 Graphs

- 4.1 Dynamic Connectivity - connected(u,v) query

1. Data Structures

1.1. Segment Tree using Pointers

```

1  /* Segment Tree implementation using pointers */
2  /* Can be adapted to Persistent Segment Tree */
3
4  struct node {
5      node *left, *right;
6      //attributes of node
7      node() {
8          //initialize attributes
9          left = NULL;
10         right = NULL;
11     }
12 };
13 void combine(node *ans, node *left, node *right){
14     //combine operation
15 }
16 void build(node *root, int l, int r){
17     if(l == r){
18         root->sum = v[l];
19         return;
20     }
21     int m = (l+r) >> 1;
22     if(!root->left) root->left = new node();
23     if(!root->right) root->right = new node();
24     build(root->left, l, m);
25     build(root->right, m+1, r);
26     combine(root, root->left, root->right);
27 }
28
29 void update(node *root, int l, int r, int idx, int val){
30     if(l == r && l == idx){
31         //do leaf operation
32         return;
33     }
34     int m = (l+r) >> 1;
35     if(idx <= m){
36         if(!root->left) root->left = new node();
37         update(root->left, l, m, idx, val);
38     }
39     else {
40         if(!root->right) root->right = new node();
41         update(root->right, m+1, r, idx, val);
42     }
43     combine(root, root->left, root->right);
44 }
45
46 node* query(node *root, int l, int r, int a, int b){
47     if(l == a && r == b){
48         return root;
49     }
50     int m = (l+r) >> 1;
51     if(b <= m){
52         if(!root->left) root->left = new node();
53         return query(root->left, l, m, a, b);
54     }
55     else if(m < a){
56         if(!root->right) root->right = new node();
57         return query(root->right, m+1, r, a, b);
58     }
59     if(!root->left) root->left = new node();
60     if(!root->right) root->right = new node();
61     node *left = query(root->left, l, m, a, m);

```

```

62 node *right = query(root->right, m+1, r, m+1, b);
63 node *ans = new node();
64 combine(ans, left, right);
65 return ans;
66 }

```

1.2. Range Update Segment Tree

```

1  /* Range update Segment Tree Implementation */
2  /* The first node (ROOT) is defined to 1 (1 - index impl) */
3  /* N is the maximum number of elements given by the statement */
4  /* Lazy can be inside node structure instead of being another structure */
5
6  #define ROOT 1
7  #define N MAX_INPUT
8
9  struct node{
10     //attributes of node
11 };
12
13 node tree[4*N];
14 node lazy[4*N];
15
16 node combine(node a, node b){
17     node res;
18     //combine operations
19     return res;
20 }
21
22 void propagate(int root, int l, int r){
23     //return if there is no update
24     //update tree using lazy node
25     if(l != r){
26         //propagate for left and right child
27     }
28     //reset lazy node
29 }
30
31 void range_update(int root, int l, int r, int a, int b, long long val){
32     if(l == a && r == b){
33         //lazy operation using val
34         return;
35     }
36
37     int m = (l+r)/2;
38
39     if(b <= m) range_update(2*root, l, m, a, b, val);
40     else if(m < a) range_update(2*root+1, m+1, r, a, b, val);
41     else {
42         range_update(2*root, l, m, a, m, val);
43         range_update(2*root+1, m+1, r, m+1, b, val);
44     }
45
46     propagate(root, l, r);
47     propagate(2*root, l, m);
48     propagate(2*root+1, m+1, r);
49     tree[root] = combine(tree[2*root], tree[2*root+1]);
50 }
51
52 node query(int root, int l, int r, int a, int b){
53     propagate(root, l, r);
54     if(l == a && r == b) return tree[root];
55
56     int m = (l+r)/2;

```

```

57 if(b <= m) return query(2*root, l, m, a, b);
58 else if(m < a) return query(2*root+1, m+1, r, a, b);
59 else {
60     node left = query(2*root, l, m, a, m);
61     node right = query(2*root+1, m+1, r, m+1, b);
62     node ans = combine(left, right);
63     return ans;
64 }
65 }

```

1.3. Range Update Binary Indexed Tree

```

1  /* Range Update Binary Indexed Tree Implementation */
2  /* Tree is 1 - index */
3  /* Point Update Binary Indexed Tree operations are used as auxiliar */
4  /* N is defined as the maximum number of elements (given by the statement) */
5
6  #define N MAX_INPUT
7
8  int bit[2][N+1];
9
10 void init(int n){
11     for(int i=1; i<=n; i++){
12         bit[0][i] = 0;
13         bit[1][i] = 0;
14     }
15 }
16
17 //auxiliar functions
18
19 void update(int *bit, int idx, int val, int n){
20     for(int i = idx; i <= n; i += i&-i){
21         bit[i] += val;
22     }
23 }
24
25 int query(int *bit, int idx){
26     int ans = 0;
27     for(int i=idx; i>0; i -= i&-i){
28         ans += bit[i];
29     }
30     return ans;
31 }
32
33 //end of auxiliar functions
34
35 void range_update(int l, int r, int val, int n){
36     update(bit[0], l, val, n);
37     update(bit[0], r+1, -val, n);
38     update(bit[1], l, val*(l-1), n);
39     update(bit[1], r+1, -val*r, n);
40 }
41
42 int prefix_query(int idx){
43     return query(bit[0], idx)*idx - query(bit[1], idx);
44 }
45
46 int range_query(int l, int r){
47     return prefix_query(r) - prefix_query(l-1);
48 }

```

2. Uncategorized

2.1. Longest Increasing Subsequence

```

1  /* Longest Increasing Subsequence Implementation */
2  /* N is defined as the maximum array size given by the statement */
3
4  #define N MAX_N
5
6  int v[N];
7  int lis[N+1];
8
9  void calculate_lis(int n){
10     for(int i=1; i<=n; i++) lis[i] = INT_MAX;
11     lis[0] = INT_MIN;
12     for(int i=0; i<n; i++){
13         int index = lower_bound(lis, lis+n+1, v[i]) - lis;
14         index--;
15         lis[index+1] = min(lis[index+1], v[i]);
16     }
17 }

```

3. Geometry

3.1. Closest Pair of Points

```

1  /* Closest Pair of Points Problem Implementation */
2  /* Divide and Conquer Approach */
3  /* Using the observation of only checking points inside min_dist x min_dist
4     from mid */
5  /* Binary search boosts search for the right border start point */
6
7  struct vec2 {
8      long long x, y;
9  };
10
11 bool cmp(vec2 a, vec2 b) {
12     return a.x < b.x || (a.x == b.x && a.y < b.y);
13 }
14
15 pair<vec2, vec2> ans;
16
17 long long solve(vector<vec2> &a) {
18     long long mid = a[a.size()/2].x;
19     int n = a.size();
20
21     vector<vec2> l;
22     vector<vec2> r;
23     int i = 0;
24     for(; i < a.size()/2; i++) l.push_back(a[i]);
25     for(; i < a.size(); i++) r.push_back(a[i]);
26
27     long long d = LLONG_MAX;
28
29     if(l.size() > 1) {
30         d = min(d, solve(l));
31     } if(r.size() > 1) {
32         d = min(d, solve(r));
33     }
34
35     a.clear();
36
37     vector<vec2> ll;

```

```

38     vector<vec2> rr;
39
40
41     int j = 0;
42     i = 0;
43     for(int k=0; k<n; k++){
44         if(i < l.size() && j < r.size()){
45             if(r[j].y <= l[i].y){
46                 if((r[j].x - mid)*(r[j].x - mid) < d) {
47                     rr.push_back(r[j]);
48                 }
49                 a.push_back(r[j++]);
50             }
51             else {
52                 if((l[i].x - mid)*(l[i].x - mid) < d) {
53                     ll.push_back(l[i]);
54                 }
55                 a.push_back(l[i++]);
56             }
57         }
58         else if(i < l.size()){
59             if((l[i].x - mid)*(l[i].x - mid) < d) {
60                 ll.push_back(l[i]);
61             }
62             a.push_back(l[i++]);
63         }
64         else {
65             if((r[j].x - mid)*(r[j].x - mid) < d) {
66                 rr.push_back(r[j]);
67             }
68             a.push_back(r[j++]);
69         }
70     }
71
72     for(int i = 0; i < ll.size(); i++) {
73
74         int ini = 0, end = rr.size()-1;
75         int j;
76         while(ini < end) {
77             j = (ini + end) / 2;
78             if((rr[j].y - ll[i].y)*(rr[j].y - ll[i].y) > d && rr[j].y < ll[i].y)
79                 ini = j+1;
80             else end = j;
81         }
82         j = ini;
83
84         for(; j < rr.size(); j++) {
85             if((rr[j].y - ll[i].y)*(rr[j].y - ll[i].y) > d) break;
86             long long cur = (ll[i].x - rr[j].x)*(ll[i].x - rr[j].x)
87                 + (ll[i].y - rr[j].y)*(ll[i].y - rr[j].y);
88             if(cur < d) {
89                 d = cur;
90                 long long cur2 = (ans.first.x - ans.second.x)*(ans.first.x -
91                     ans.second.x)
92                     + (ans.first.y - ans.second.y)*(ans.first.y - ans.second.y);
93                 if(cur < cur2)
94                     ans = { ll[i], rr[j] };
95             }
96         }
97     }
98     return d;
99 }

```

3.2. Convex Hull - Monotone Chain Algorithm

```

1  /* Convex Hull - Monotone Chain */
2  /* Generates Upper and Lower Hull */
3  /* It is needed to give array of points ordered by <x,y> */
4
5  vector < pair<int, int> > upper, lower;
6  vector< pair<int, int> > hull;
7
8
9  int cross(pair<int, int> & a, pair<int, int> & b, pair<int, int> & c){
10     pair<int, int> vec1(b.ff - a.ff, b.ss - a.ss);
11     pair<int, int> vec2(c.ff - b.ff, c.ss - b.ss);
12     return vec1.ff*vec2.ss - vec1.ss*vec2.ff;
13 }
14
15 void calculate_upper(vector< pair<int, int> > & p){
16     for(int i=0; i<p.size(); i++){
17         while(upper.size() >= 2 && cross(upper[upper.size()-2],
18             upper.back(), p[i]) >= 0){
19             upper.pop_back();
20         }
21         upper.push_back(p[i]);
22     }
23 }
24 void calculate_lower(vector< pair<int, int> > & p){
25     for(int i=0; i<p.size(); i++){
26         while(lower.size() >= 2 && cross(lower[lower.size()-2],
27             lower.back(), p[i]) <= 0){
28             lower.pop_back();
29         }
30         lower.push_back(p[i]);
31     }
32 }
33 void merge_hull(){
34     for(int i=0; i<upper.size(); i++) hull.push_back(upper[i]);
35     for(int i=lower.size()-2; i>0; i--) hull.push_back(lower[i]);
36 }

```

3.3. Shoelace Formula for Polygon Area

```

1  /* Shoelace formula */
2  /* Calculate area of convex polygon */
3  /* Points given in clockwise/counterclockwise order */
4
5  int cross(pair<int, int> & a, pair<int, int> & b){
6     return a.ff*b.ss - a.ss*b.ff;
7 }
8
9  int shoelace(vector< pair<int, int> > & p){
10     int area = 0;
11     for(int i=0; i<hull.size(); i++){
12         area += cross(hull[i], hull[(i+1)%hull.size()]);
13     }
14     return abs(area/2);
15 }

```

4. Graphs

4.1. Dynammic Connectivity - connected(u,v) query

```

1  /* Dynammic Connectivity Implementation */

```

```

2  /* Uses Divide and Conquer Offline approach */
3  /* Able to answer if two vertex <u,v> are connected */
4  /* No multi-edges allowed */
5  /* DSU + Rollback is used to backtrack merges */
6  /* N is defined as the maximum graph size given by input */
7
8  #define N MAX_INPUT
9
10 int uf[N];
11 int sz[N];
12
13 struct event{
14     int op, u, v, l, r;
15     event() {}
16     event(int o, int a, int b, int x, int y) : op(o), u(a), v(b), l(x), r(y) {}
17 };
18
19 map< pair<int, int>, int > edge_to_l;
20 stack< pair<int*, int> > hist;
21 vector<event> events;
22
23 int init(int n){
24     for(int i=0; i<=n; i++){
25         uf[i] = i;
26         sz[i] = 1;
27     }
28 }
29
30 int find(int u){
31     if(uf[u] == u) return u;
32     else return find(uf[u]);
33 }
34
35 void merge(int u, int v){
36     int a = find(u);
37     int b = find(v);
38     if(a == b) return;
39     if(sz[a] < sz[b]){
40         hist.push(make_pair(&uf[a], uf[a]));
41         uf[a] = b;
42         hist.push(make_pair(&sz[b], sz[b]));
43         sz[b] += sz[a];
44     }
45     else {
46         hist.push(make_pair(&uf[b], uf[b]));
47         hist.push(make_pair(&sz[a], sz[a]));
48         uf[b] = a;
49         sz[a] += sz[b];
50     }
51 }
52
53 int snap(){
54     return hist.size();
55 }
56
57 void rollback(int t){
58     while(hist.size() > t){
59         pair<int*, int> aux = hist.top();
60         hist.pop();
61         *aux.first = aux.second;
62     }
63 }
64
65 void solve(int l, int r){
66     if(l == r){

```

```

67     if(events[l].op == 2){
68         if(find(events[l].u) == find(events[l].v)) cout << "YES" << endl;
69         else cout << "NO" << endl;
70     }
71     return;
72 }
73
74 int m = (l+r)/2;
75 //doing for [L,m]
76 int t = snap();
77 for(int i=l; i<=r; i++){
78     if(events[i].op == 0 || events[i].op == 1){
79         if(events[i].l <= l && m <= events[i].r) merge(events[i].u,
80             events[i].v);
81     }
82 }
83 solve(l, m);
84 rollback(t);
85
86 //doing for [m+1, R]
87 t = snap();
88 for(int i=l; i<=r; i++){
89     if(events[i].op == 0 || events[i].op == 1){
90         if(events[i].l <= m+1 && r <= events[i].r) merge(events[i].u,
91             events[i].v);
92     }
93 }
94 solve(m+1, r);
95 rollback(t);
96 }
97
98 void offline_process(){
99     int n, q;
100     cin >> n >> q; //number of vertex and queries
101     init(n);
102     for(int i=0; i<q; i++){
103         string op;
104         int u,v;
105         cin >> op >> u >> v; //add, remove or query for u,v
106         if(u > v) swap(u,v);
107         if(op == "add"){
108             events.push_back(event(0, u, v, i, -1));
109             edge_to_l[make_pair(u,v)] = i;
110         }
111         else if(op == "rem"){
112             int l = edge_to_l[make_pair(u,v)];
113             events.push_back(event(l, u, v, l, i));
114             events[l].r = i;
115         }
116         else if(op == "conn"){
117             events.push_back(event(2, u, v, -1, -1));
118         }
119     }
120     for(int i=0; i<q; i++){
121         if(events[i].op == 0){
122             if(events[i].r == -1){
123                 events[i].r = events.size();
124                 events.push_back(event(l, events[i].u, events[i].v, events[i].l,
125                     events[i].r));
126             }
127         }
128     }
129 }

```

5. Math and Number Theory

5.1. Binomial Coefficient DP

```

1  /* Dynamic Programming for Binomial Coefficient Calculation */
2  /* Using Stiefel Rule C(n, k) = C(n-1, k) + C(n-1, k-1) */
3
4  int binomial(int n ,int k){
5      int c[n+10][k + 10];
6      memset(c, 0 , sizeof c);
7      c[0][0] = 1;
8      for(int i = 1; i<=n; i++){
9          for(int j = min(i, k); j>0; j--){
10             c[i][j] = c[i-1][j] + c[i-1][j-1];
11         }
12     }
13     return c[n][k];
14 }

```

5.2. Erathosthenes Sieve + Logn Prime Factorization

```

1  /* Erathosthenes Sieve Implementation */
2  /* Calculate primes from 2 to N */
3  /* lf[i] stores the lowest prime factor of i(logn factorization) */
4
5  bitset<N> prime;
6  int lf[N];
7
8  void run_sieve(int n){
9      for(int i=0; i<=n; i++) lf[i] = i;
10     prime.set();
11     prime[0] = false;
12     prime[1] = false;
13     for(int p = 2; p*p <= n; p++){
14         if(prime[p]){
15             for(int i=p*p; i<=n; i+=p){
16                 prime[i] = false;
17                 lf[i] = min(lf[i], p);
18             }
19         }
20     }
21 }

```

5.3. Matrix Exponentiation

```

1  /* Matrix Exponentiation Implementation */
2
3  typedef vector< vector<int> > Matrix;
4
5  Matrix operator *(const Matrix & a, const Matrix & b){
6      Matrix c(a.size(), vector<int>(b[0].size()));
7      for(int i = 0; i<a.size(); i++){
8          for(int j = 0; j<b[0].size(); j++){
9              for(int k = 0; k<b.size(); k++){
10                 c[i][j] += (a[i][k]*b[k][j]);
11             }
12         }
13     }
14     return c;
15 }
16
17 Matrix exp(Matrix & a, int k){
18     if(k == 1) return a;

```

```

19 Matrix c = exp(a, k/2);
20 c = c*c;
21 if(k%2) c = c*a;
22 return c;
23 }

```

6. String Algorithms

6.1. KMP Failure Function + String Matching

```

1  /* Knuth - Morris - Pratt Algorithm */
2  /* Failure Function for String Matching */
3
4
5  int pi[N];
6  string s, t;
7
8  void prefix(int n) {
9      pi[0] = 0;
10     for(int i = 1; i < n; i++) {
11         pi[i] = pi[i-1];
12         while(pi[i] > 0 && t[i] != t[pi[i]]) pi[i] = pi[pi[i]-1];
13         if(t[i] == t[pi[i]]) pi[i]++;
14     }
15 }
16
17 void matching(int n){
18     int j = 0;
19     for(int i=0; i<n; i++){
20         while(j > 0 && s[i] != t[j]) j = pi[j-1];
21         if(s[i] == t[j]) j++;
22         if(j == t.size()){
23             cout << "match in " << j-t.size()+1 << endl;
24             j = pi[j-1];
25         }
26     }
27 }

```

6.2. Z-Function

```

1  /* Z-function */
2  /* Calculate the size K of the largest substring which is a prefix */
3
4  vector<int> z;
5
6  void make(string s){
7      int n = s.size();
8      z.resize(n);
9      z[0] = 0;
10     int l = 0, r = 0;
11     for(int i=1; i<n; i++){
12         if(i > r){
13             l = i;
14             r = i;
15         }
16         z[i] = min(z[i-l], r-i+1);
17         while(i + z[i] < n && s[i + z[i]] == s[z[i]]) z[i]++;
18         if(i + z[i] > r){
19             l = i;
20             r = i + z[i]-1;
21         }
22     }
23 }

```

6.3. Rolling Hash

```

1  /* Rolling Hash Implementation */
2  /* Uses 1-indexed string */
3
4  long long BASE = 137
5  long long PRIME = (int)1e9+7;
6
7  long long hash[N+1];
8  long long base[N+1];
9  long long invBase[N+1];
10
11 long long expo(long long a, long long k){
12     if(k == 0) return 1LL;
13     else if(k == 1) return a;
14     long long aux = expo(a, k/2);
15     aux %= PRIME;
16     aux *= aux;
17     aux %= PRIME;
18     if(k%2) aux *= a;
19     aux %= PRIME;
20     return aux;
21 }
22
23 void calculate(string a){
24     base[0] = 1;
25     invBase[0] = 1;
26     hash[0] = 0;
27     for(int i=1; i<=a.size(); i++){
28         hash[i] += BASE*hash[i-1] + a[i-1];
29         hash[i] %= PRIME;
30         base[i] = base[i-1]*BASE;
31         base[i] %= PRIME;
32         invBase[i] = expo(base[i], PRIME-2);
33     }
34 }
35
36 long long range_hash(int i, int j){
37     return ((h[j] - h[i-1])*invBase[j-i+1])%PRIME;
38 }

```

6.4. Suffix Array + Linear Sort

```

1  /* Suffix Array using Counting Sort Implementation */
2  /* rnk is inverse of sa array */
3  /* aux arrays are needed for sorting step */
4  /* inverse sorting (using rotating arrays and blocks of power of 2) */
5  /* rmq data structure needed for calculating lcp of two non adjacent
   suffixes sorted */
6
7  int rnk[N], tmp[N], sa[N], sa_aux[N], lcp[N];
8  int block=0, n;
9  string s;
10
11 bool suffixcmp(int i, int j){
12     if(rnk[i] != rnk[j]) return rnk[i] < rnk[j];
13     i+=block, j+=block;
14     i%=n;
15     j%=n;
16     return rnk[i] < rnk[j];
17 }
18
19 void suffixSort(int MAX_VAL){
20     for(int i=0; i<=MAX_VAL; i++) tmp[i] = 0;

```

```

21 for(int i=0; i<n; i++) tmp[rnk[i]]++;
22 for(int i=1; i<=MAX_VAL; i++) tmp[i] += tmp[i-1];
23 for(int i = n-1; i>=0; i--){
24     int aux = sa[i]-block;
25     aux%=n;
26     if(aux < 0) aux+=n;
27     sa_aux[--tmp[rnk[aux]]] = aux;
28 }
29 for(int i=0; i<n; i++) sa[i] = sa_aux[i];
30 tmp[0] = 0;
31 for(int i=1; i<n; i++) tmp[i] = tmp[i-1] + suffixcmp(sa[i-1], sa[i]);
32 for(int i=0; i<n; i++) rnk[sa[i]] = tmp[i];
33 }
34
35 void build_sa(){
36     s+='\0';
37     n++;
38     for(int i=0; i<n; i++){
39         sa[i] = i;
40         rnk[i] = s[i];
41         tmp[i] = 0;
42     }
43     suffixSort(256);
44     block = 1;
45     while(tmp[n-1] != n-1){
46         suffixSort(tmp[n-1]);
47         block*=2;
48     }
49     for(int i=0; i<n-1; i++) sa[i] = sa[i+1];
50     n--;
51     tmp[0] = 0;
52     for(int i=1; i<n; i++) tmp[i] = tmp[i-1] + suffixcmp(sa[i-1], sa[i]);
53     for(int i=0; i<n; i++) rnk[sa[i]] = tmp[i];
54 }
55
56 void calculate_lcp(){
57     int last = 0;
58     for(int i=0; i<n; i++){
59         if(rnk[i] == n-1) continue;
60         int x = rnk[i];
61         lcp[x] = max(0, last-1);
62         while(sa[x] + lcp[x] < n && sa[x+1] + lcp[x] < n && s[sa[x]+lcp[x]] ==
63             s[sa[x+1]+lcp[x]]){
64             lcp[x]++;
65         }
66         last = lcp[x];
67     }
68 }
69
70 int lcp(int x, int y){
71     if(x == y) return n - x;
72     if(rnk[x] > rnk[y]) swap(x,y);
73     return rmq(rnk[x], rnk[y]-1); //segtree

```