Competitive Programming Algorithms and Topics

Ubiratan Neto

1. Data Structures

1.1. Segment Tree using Pointers

```
/* Segment Tree implementation using pointers */
/* Can be adapted to Persistent Segment Tree */

struct node {
  node *left, *right;
  //attributes of node
  node() {
    //initialize attributes
    left = NULL;
    right = NULL;
  }
};
void combine(node *ans, node *left, node *right){
  //combine operation
}
void build(node *root, int l, int r){
  if(l == r){
    root->sum = v[l];
    return;
  }
  int m = (l+r) >> 1;
  if(!root->left) root->left = new node();
  if(!root->right) root->right = new node();
  build(root->left, l, m);
  build(root->right, m+1, r);
  combine(root, root->left, root->right);
}

void update(node *root, int l, int r, int idx, int val){
  if(l == r && l == idx){
    //do leaf operation
    return;
  }
  int m = (l+r) >> 1;
  if(idx <= m){
    if(!root->left) root->left = new node();
    update(root->left, l, m, idx, val);
  }
  else {
    if(!root->right) root->right = new node();
    update(root->right, m+1, r, idx, val);
  }
  combine(root, root->left, root->right);
}

node* query(node *root, int l, int r, int a, int b){
  if(l == a && r == b){
    return root;
  }
  int m = (l+r) >> 1;
  if(b <= m){
    if(!root->left) root->left = new node();
    return query(root->left, l, m, a, b);
  }
  else if(m < a){
    if(!root->right) root->right = new node();
    return query(root->right, m+1, r, a, b);
  }
  if(!root->left) root->left = new node();
  if(!root->right) root->right = new node();
  node *left = query(root->left, l,m,a,m);
```

```
  node *right = query(root->right, m+1, r, m+1, b);
  node *ans = new node();
  combine(ans, left, right);
  return ans;
}
```

1.2. Range Update Segment Tree

```
/* Range update Segment Tree Implementation */
/* The first node (ROOT) is defined to 1 (1 - index impl) */
/* N is the maximum number of elements given by the statement */
/* Lazy can be inside node structure instead of being another structure */

#define ROOT 1
#define N MAX_INPUT

struct node{
  //attributes of node
};

node tree[4*N];
node lazy[4*N];

node combine(node a, node b){
  node res;
  //combine operations
  return res;
}

void propagate(int root, int l , int r){
  //return if there is no update
  //update tree using lazy node
  if(l != r){
    //propagate for left and right child
  }
  //reset lazy node
}

void range_update(int root, int l, int r, int a, int b, long long val){
  if(l == a && r == b){
    //lazy operation using val
    return;
  }

  int m = (l+r)/2;

  if(b <= m) range_update(2*root, l, m, a, b, val);
  else if(m < a) range_update(2*root+1, m+1, r, a, b, val);
  else {
    range_update(2*root, l, m, a, m, val);
    range_update(2*root+1, m+1, r, m+1, b, val);
  }

  propagate(root, l , r);
  propagate(2*root, l, m);
  propagate(2*root+1, m+1, r);
  tree[root] = combine(tree[2*root], tree[2*root+1]);
}

node query(int root, int l, int r, int a, int b){
  propagate(root, l, r);
  if(l == a && r == b) return tree[root];

  int m = (l+r)/2;
  if(b <= m) return query(2*root, l, m, a, b);
  else if(m < a) return query(2*root+1, m+1, r, a, b);
  else {
    node left = query(2*root, l, m, a, m);
    node right = query(2*root+1, m+1, r, m+1, b);
    node ans = combine(left, right);
    return ans;
  }
}
```

1.3. Range Update Binary Indexed Tree

```
/* Range Update Binary Indexed Tree Implementation */
/* Tree is 1 - index */
/* Point Update Binary Indexed Tree operations are used as auxiliar */
/* N is defined as the maximum number of elements (given by the statement) */

#define N MAX_INPUT

int bit[2][N+1];

void init(int n){
  for(int i=1; i<=n; i++){
    bit[0][i] = 0;
    bit[1][i] = 0;
  }
}

//auxiliar functions

void update(int *bit, int idx, int val, int n){
  for(int i = idx; i <= n; i += i&-i){
    bit[i]+=val;
  }
}

int query(int *bit, int idx){
  int ans = 0;
  for(int i=idx; i>0; i -= i&-i){
    ans += bit[i];
  }
  return ans;
}

//end of auxiliar functions

void range_update(int l, int r, int val, int n){
  update(bit[0], l, val, n);
  update(bit[0], r+1, -val, n);
  update(bit[1], l, val*(l-1), n);
  update(bit[1], r+1, -val*r, n);
}

int prefix_query(int idx){
  return query(bit[0],idx)*idx - query(bit[1], idx);
}

int range_query(int l, int r){
  return prefix_query(r) - prefix_query(l-1);
}
```

2. Uncategorized

2.1. Longest Increasing Subsequence

```
/* Longest Increasing Subsequence Implementation */
/* N is defined as the maximum array size given by the statement */

#define N MAX_N

int v[N];
int lis[N+1];

void calculate_lis(int n){
  for(int i=1; i<=n; i++) lis[i] = INT_MAX;
  lis[0] = INT_MIN;
  for(int i=0; i<n; i++){
    int index = lower_bound(lis, lis+n+1, v[i]) - lis;
    index--;
    lis[index+1] = min(lis[index+1], v[i]);
  }
}
```

3. Geometry

3.1. Closest Pair of Points

```
/* Closest Pair of Points Problem Implementation */
/* Divide and Conquer Approach */
/* Using the observation of only checking points inside min_dist x min_dist
    from mid */
/* Binary search boosts search for the right border start point */

struct vec2 {
  long long x, y;
};

bool cmp(vec2 a, vec2 b) {
  return a.x < b.x || (a.x == b.x && a.y < b.y);
}

pair<vec2, vec2> ans;

long long solve(vector<vec2> &a) {

  long long mid = a[a.size()/2].x;
  int n = a.size();

  vector<vec2> l;
  vector<vec2> r;
  int i = 0;
  for(; i < a.size()/2; i++) l.push_back(a[i]);
  for(; i < a.size(); i++) r.push_back(a[i]);

  long long d = LLONG_MAX;

  if(l.size() > 1) {
    d = min(d, solve(l));
  } if(r.size() > 1) {
    d = min(d, solve(r));
  }

  a.clear();

  vector<vec2> ll;
  vector<vec2> rr;


  int j = 0;
  i = 0;
  for(int k=0; k<n; k++){
    if(i < l.size() && j < r.size()){
      if(r[j].y <= l[i].y){
        if((r[j].x - mid)*(r[j].x - mid) < d) {
          rr.push_back(r[j]);
        }
        a.push_back(r[j++]);
      }
      else {
        if((l[i].x - mid)*(l[i].x - mid) < d) {
          ll.push_back(l[i]);
        }
        a.push_back(l[i++]);
      }
    }
    else if(i < l.size()){
      if((l[i].x - mid)*(l[i].x - mid) < d) {
        ll.push_back(l[i]);
      }
      a.push_back(l[i++]);
    }
    else {
      if((r[j].x - mid)*(r[j].x - mid) < d) {
          rr.push_back(r[j]);
      }
        a.push_back(r[j++]);
    }
  }

  for(int i = 0; i < ll.size(); i++) {

    int ini = 0, end = rr.size()-1;
    int j;
    while(ini < end) {
      j = (ini + end) / 2;
      if((rr[j].y - ll[i].y)*(rr[j].y - ll[i].y) > d && rr[j].y < ll[i].y)
        ini = j+1;
      else end = j;
    }
    j = ini;

    for(; j < rr.size(); j++) {
      if((rr[j].y - ll[i].y)*(rr[j].y - ll[i].y) > d) break;
      long long cur =  (ll[i].x - rr[j].x)*(ll[i].x - rr[j].x)
              +(ll[i].y - rr[j].y)*(ll[i].y - rr[j].y);
      if(cur < d) {
        d = cur;
        long long cur2 =  (ans.first.x - ans.second.x)*(ans.first.x -
    ans.second.x)
              +(ans.first.y - ans.second.y)*(ans.first.y - ans.second.y);
        if(cur < cur2)
          ans = { ll[i], rr[j] };
      }
    }
  }
  return d;
}
```

4. Graphs

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

```cpp
/* Dynammic Connectivity Implementation */
/* Uses Divide and Conquer Offline approach */
/* Able to answer if two vertex <u,v> are connected */
/* No multi-edges allowed */
/* DSU + Rollback is used to backtrack merges */
/* N is defined as the maximum graph size given by input */

#define N MAX_INPUT

int uf[N];
int sz[N];

struct event{
  int op, u, v, l, r;
  event() {}
  event(int o, int a, int b, int x, int y) : op(o), u(a), v(b), l(x), r(y) {}
};

map< pair<int, int>, int > edge_to_l;
stack< pair<int*,int> > hist;
vector<event> events;

int init(int n){
  for(int i=0; i<=n; i++){
    uf[i] = i;
    sz[i] = 1;
  }
}

int find(int u){
  if(uf[u] == u) return u;
  else return find(uf[u]);
}

void merge(int u, int v){
  int a = find(u);
  int b = find(v);
  if(a == b) return;
  if(sz[a] < sz[b]){
    hist.push(make_pair(&uf[a], uf[a]));
    uf[a] = b;
    hist.push(make_pair(&sz[b], sz[b]));
    sz[b]+= sz[a];
  }
  else {
    hist.push(make_pair(&uf[b], uf[b]));
    hist.push(make_pair(&sz[a], sz[a]));
    uf[b] = a;
    sz[a]+=sz[b];
  }
}

int snap(){
  return hist.size();
}

void rollback(int t){
  while(hist.size() > t){
    pair<int*, int> aux = hist.top();
    hist.pop();
    *aux.first = aux.second;
  }
}

void solve(int l, int r){
  if(l == r){
    if(events[l].op == 2){
      if(find(events[l].u) == find(events[l].v)) cout << "YES" << endl;
      else cout << "NO" << endl;
    }
    return;
  }

  int m = (l+r)/2;
  //doing for [L,m]
  int t = snap();
  for(int i=l; i<=r; i++){
    if(events[i].op == 0 || events[i].op == 1){
      if(events[i].l <= l && m <= events[i].r) merge(events[i].u,
    events[i].v);
    }
  }
  solve(l, m);
  rollback(t);

  //doing for [m+1, R]
  t = snap();
  for(int i=l; i<=r; i++){
    if(events[i].op == 0 || events[i].op == 1){
      if(events[i].l <= m+1 && r <= events[i].r) merge(events[i].u,
    events[i].v);
    }
  }
  solve(m+1, r);
  rollback(t);
}

void offline_process(){
  int n, q;
  cin >> n >> q; //number of vertex and queries
  init(n);
  for(int i=0; i<q; i++){
    string op;
    int u,v;
    cin >> op >> u >> v; //add, remove or query for u,v
    if(u > v) swap(u,v);
    if(op == "add"){
      events.push_back(event(0, u, v, i, -1));
      edge_to_l[make_pair(u,v)] = i;
    }
    else if(op == "rem"){
      int l = edge_to_l[make_pair(u,v)];
      events.push_back(event(1, u, v, l, i));
      events[l].r = i;
    }
    else if(op == "conn"){
      events.push_back(event(2, u, v, -1, -1));
    }
  }
  for(int i=0; i<q; i++){
    if(events[i].op == 0){
      if(events[i].r == -1){
        events[i].r = events.size();
        events.push_back(event(1, events[i].u, events[i].v, events[i].l,
    events[i].r));
      }
```

```
        }
    }
}
```