Contents

1 Data Structures

1.1 Sets

1.1.1 Union-Find Disjoint Sets

build $O(n)$

```
int un[MAXV];
int rnk[MAXV];

void setUn(){
  for(int i = 0; i < MAXV; ++i)
    un[i] = i;
}
```

Initializes each value as member of its own set

build $\approx O(1)$

```
int find(int a){
  return un[a] == a? a : un[a] = find(un[a]);
}
```

Tells the set a belongs to.

merge $\approx O(1)$

```
bool merge(int a, int b){
  x = find(a);
  y = find(b);
  if(x != y){
    if (rnk[x] > rnk[y]) un[y] = x;
    else {
      un[x] = y;
      if (rnk[x] == rnk[y]) rnk[y]++;
    }
    return true;
  }
  return false;
}
```

Merge the sets in which a and b belong to.

1.2 Trees

1.2.1 Segment Tree

build $O(n)$

```
ll arr[MAXN];
ll tree[MAXN*4];

void build(int node, int l, int r){
  if(l > r) return;
  if(l == r){
    tree[node] = arr[l];
    return;
  }
  build(node*2, l, (l+r)/2);
  build(node*2+1, 1+(l+r)/2, r);
  tree[node] = tree[node*2] + tree[node*2+1];
}
```

Stores the values of *arr* in *tree*, each node stores the operation on a different interval of values. Root of *tree* is 1.

query $O(\log n)$

```
ll query(int node, int l, int r, int a, int b){
  if(l > r || l > b || r < a) return 0;
  if(l >= a && r <= b){
    return tree[node];
  }
  ll temp1 = query(node*2, l, (l+r)/2, a, b);
  ll temp2 = query(node*2+1, 1+(l+r)/2, r, a, b);
  return temp1 + temp2;
}
```

Returns the given operation on the range $[a, b]$.

update $O(n)$

```
void update(int node, int l, int r, int a, int b, ll p){
  if(l > r || l > b || r < a) return;
  if(l == r){
    tree[node] += p;
    return;
  }
  update(node*2, l, (l+r)/2, a, b, p);
  update(node*2+1, 1+(l+r)/2, r, a, b, p);
  tree[node] = tree[node*2] + tree[node*2+1];
}
```

Increments every value in the range $[a, b]$ in p units, can be modified to change all values in the range to p by changing increment operator by assignation.

propagate $O(1)$

```
void prop(int node, int l, int r){
  if(lazy[node]){
    tree[node] += lazy[node];
    if(l!=r){
      lazy[node*2] += lazy[node];
      lazy[node*2+1] += lazy[node];
    }
    lazy[node] = 0;
  }
}
```

Updates the value on *node* and propagates the lazy value to its children.

lazy update $O(\log n)$

```
void update(int node, int l, int r, int a, int b, ll p){
  prop(node, l, r);
  if(l > r || l > b || r < a) return;
  if(l >= a && r <= b){
    tree[node] += p;
    if(l!=r){
      lazy[node*2] += p;
      lazy[node*2+1] += p;
    }
    return;
  }
  update(node*2, l, (l+r)/2, a, b, p);
  update(node*2+1, 1+(l+r)/2, r, a, b, p);
  tree[node] = min(tree[node*2], tree[node*2+1]);
}
```

Increments every value in the range $[a, b]$ in p units, lazy values are stored in *lazy*, updates on demand.

lazy query $O(\log n)$

```
ll query(int node, int l, int r, int a, int b){
  if(l > r || l > b || r < a) return INF;
  prop(node, left, right);
  if(l >= a && r <= b){
    return tree[node];
  }
  ll temp1 = query(node*2, l, (l+r)/2, a, b);
  ll temp2 = query(node*2+1, 1+(l+r)/2, r, a, b);
  return min(temp1, temp2);
}
```

Computes the value of the operation in the range $[a, b]$, updates on demand.

2 Graphs

2.1 MST

2.1.1 Kruskal

Kruskal $O(|E| \log |V|)$

```
vector< tuple<double, int, int> > edges;

int MST(int t){
  setUn();
  sort(edges.begin(), edges.end());
  int cost = 0;
  for(int i = 0; i < (int)edges.size(); ++i){
    bool f = merge(get<1>(edges[i]), get<2>(edges[i]));
    cost += (get<0>(edges[i])) * f;
  }
  return cost;
}
```

Computes the Minimum Spanning Tree of a graph with $E = edges$.

2.2 SSSP

2.2.1 Dijkstra

Dijkstra $O(|E| + |V| \log |V|)$

```
int d[MAXV];

void dijkstra(int s, int e){
  priority_queue<ii> q;
  q.push({0,s});
  d[s] = 0;
  while(!q.empty()){
    int w = -q.top().first;
    int u = q.top().second;
    if(u == e) return w;
    q.pop();
    if(w > d[u]) continue;
    for(auto &t: graph[u]){
      int nw = t.first + w;
      init v = t.second;
      if(nw < d[v]){
        d[v] = nw;
        q.push({-nw, v});
      }
    }
  }
  return INF;
}
```

Computes the shortest distance $d[u]$ from vertex s to every vertex u, stops after finding min distance to e.

3 Math

3.1 Series

3.1.1 Aritmetic Progression

$$S_n = a_1 + (a_1 + d) + (a_1 + 2d) + \cdots + (a_1 + (n-1)d)$$
$$S_n = \frac{n}{2}(2a_1 + (n-1)d)$$