noiref

ngmh

December 2019

Contents

1 Data Structures

Data Structure	Precomputation / Update	Query	Memory	Notes
Prefix Sum	O(N) / X	O(1)	O(N)	Associative Functions (+, XOR)
Sparse Table	O(N log N) / X	O(1)	O(N log N)	Non-Associative Functions (max, gcd)
Fenwick Tree	X / O(log N)	O(log N)	O(N)	Prefix Sum with Updates
Segment Tree	X / O(log N)	O(log N)	O(4N)	Allows more Information

Table 1: Quick Summary of Data Structures

1.1 Prefix Sums

Prefix Sums rely on the Principle of Inclusion and Exclusion. By adding and subtracting the correct prefixes, we can determine the answer for any subarray.
This idea can be extended to multiple dimensions as well, but beyond 2 it gets slightly cancerous.
Query: O(1)
Update: X

1.1.1 1D

```
//Query - 1-Indexed
int query(int s, int e){
    return ps[e]-ps[s-1];
}

//Precomputation
for(int i = 1; i <= n; i++) ps[i] = ps[i-1]+a[i];
```

1.1.2 2D

```
//Query - 1-Indexed
int query(int x1, int y1, int x2, int y2){
    return ps[x2][y2]-ps[x1-1][y2]-ps[x2][y1-1]+ps[x1-1][y1-1];
}

//Precomputation
for(int i = 1; i <= r; i++){
    for(int j = 1; j <= c; j++){
        ps[i][j] = ps[i-1][j]+ps[i][j-1]-ps[i-1][j-1]+a[i][j];
    }
}
```

1.2 Sparse Table

Sparse Tables to me, are Prefix Sums on steroids. Instead of using prefixes, we use subarrays with sizes which are powers of 2. Then, any query will need at most 2 of the calculated subarrays.
Query: O(1)
Update: X

1.2.1 1D

```
//Query
int query(int l, int r){
    r++;
    int p = 31-__builtin_clz(r-l);
    return __gcd(sp[p][l], sp[p][r-(1<<p)]);
```

```
}

//Precomputation
h = floor(log2(n));
for(int i = 0; i < n; i++) sp[0][i] = a[i];
for(int i = 1; i <= h; i++){
    for(int j = 0; j+(1<<i) <= n; j++){
        sp[i][j] = __gcd(sp[i-1][j], sp[i-1][j+(1<<(i-1))]);
    }
}
```

1.2.2 2D

```
Not Implemented Yet!
```

1.3 Fenwick Trees

Fenwick Trees essentially act the same as prefix sums, except that they can perform updates. However, the complexity of code varies depending on what kind of queries and updates are needed. One cool use case is for range maximum, but only works when updates are strictly non-decreasing.
Query: O(log N)
Update: O(log N)

1.3.1 Point Update, Range Query

```
int ls(int x){ return (x)&(-x); }

void pu(int i, int v){
    for(; i <= n; i += ls(i)) fw[i] += v;
}
int pq(int i){
    int t = 0;
    for(; i; i -= ls(i)) t += fw[i];
    return t;
}
int rq(int s, int e){
    return pq(e)-pq(s-1);
}
```

1.3.2 Range Update, Point Query

```
int ls(int x){ return (x)&(-x); }

//PURQ Code (PU, PQ)

void ru(int s, int e, int v){
    pu(s, v);
    pu(e+1, -v);
}
```

1.3.3 Range Update, Range Query

```
int ls(int x){ return (x)&(-x); }

//PURQ Code (PU, PQ)
//Modify functions:
//int pu(*tree...

void ru(int s, int e, int v){
    pu(fw1, s, v);
    pu(fw1, e+1, -v);
    pu(fw2, s, -v*(s-1));
    pu(fw2, e+1, v*e);
}
int ps(int i){
    return pq(fw1, i)*i+pq(fw2, i);
}
int rq(int s, int e){
    return ps(e)-ps(s-1);
}
```

1.4 Segment Trees

Segment Trees need more memory and code to implement, but are much more powerful, since more information can be stored in each node. They can be used for anything really, e.g. finding how many elements are larger than K, Maxsum.
Query: O(log N)
Update: O(log N)

1.4.1 1D

```
struct node {
    int s, e, m, v;
    node *l, *r;
    node(int _s, int _e){
        s = _s; e = _e; m = (s+e)/2; v = 0;
        if(s != e){
            l = new node(s, m);
            r = new node(m+1, e);
        }
    }
    void pu(int x, int y){
        if(s == e){ v = y; return; }
        if(x <= m) l->pu(x, y);
        if(x > m) r->pu(x, y);
        v = min(l->v, r->v);
    }
    int rq(int x, int y){
        if(s == x && e == y) return v;
        if(y <= m) return l->rq(x, y);
        if(x > m) return r->rq(x, y);
        return min(l->rq(x, m), r->rq(m+1, y));
    }
} *root;
```

1.4.2 Lazy Propagation - Range Add

```
int pu(){
    if(s == e){ v += lazy; lazy = 0; return v; }
    v += lazy;
    l->lazy += lazy; r->lazy += lazy;
    lazy = 0;
    return v;
}

void ru(int x, int y, int z){
    if(s == x && e == y){ lazy += z; return; }
    if(y <= m) l->ru(x, y, z);
    else if(x > m) r->ru(x, y, z);
    else l->ru(x, m, z), r->ru(m+1, y, z);
    v = max(l->pu(), r->pu());
}

int rq(int x, int y){
    pu();
    if(s == x && e == y) return pu();
    if(y <= m) return l->rq(x, y);
    if(x > m) return r->rq(x, y);
    return max(l->rq(x, m), r->rq(m+1, y));
}
```

1.4.3 Lazy Propagation - Range Add + Set

```
Not Implemented Yet!
```

1.4.4 Maxsum Tree

```
struct node {
    ll s, e, m, ps, ss, ms, ts;
    node *l, *r;
    node(ll _s, ll _e){
        s = _s; e = _e; m = (s+e)/2; ps = ss = ms = ts = 0;
        if(s != e){
            l = new node(s, m);
            r = new node(m+1, e);
        }
    }
    void pu(ll x, ll y){
        if(s == e){ ps = ss = ms = ts = y; return; }
        if(x <= m) l->pu(x, y);
        if(x > m) r->pu(x, y);
        //New Prefix Max - Left Prefix, Left + Right Prefix
        ps = max(l->ps, l->ts+r->ps);
        //New Suffix Max - Right Suffix, Left Suffix + Right
        ss = max(r->ss, r->ts+l->ss);
        //Total Sum - Left + Right
        ts = l->ts+r->ts;
        //Maxsum - Left Suffix + Right Prefix, Left, Right,
        //         Total, Left Maxsum, Right Maxsum
        ms = max({l->ss+r->ps, ps, ss, ts, l->ms, r->ms});
    }
    ll ans(){
        return ms;
```

```
    }
} *root;
```

1.4.5 Order Statistic Tree - More than K

```
struct node {
    int s, e, m, it;
    vector<int> v;
    node *l, *r;
    node(int _s, int _e){
        s = _s; e = _e; m = (s+e)/2;
        if(s != e){
            l = new node(s, m);
            r = new node(m+1, e);
        }
    }
    void pu(int x, int y){
        if(s == e){ v.push_back(y); return; }
        if(x > m) r->pu(x, y);
        if(x <= m) l->pu(x, y);
        v.push_back(y);
    }
    void sorty(){
        if(s == e) return;
        l->sorty();
        r->sorty();
        sort(v.begin(), v.end());
    }
    int rq(int x, int y, int k){
        if(x > y) return 0;
        if(s == x && e == y) return upper_bound(v.begin(), v.end(), k)-v.begin();
        if(x > m) return r->rq(x, y, k);
        if(y <= m) return l->rq(x, y, k);
        return l->rq(x, m, k)+r->rq(m+1, y, k);
        }
    int mrq(int x, int y){
        if(x > y) return 0;
        if(s == x && e == y){ auto it = v.end(); it--; return *it; }
        if(x > m) return r->mrq(x, y);
        if(y <= m) return l->mrq(x, y);
        return max(l->mrq(x, m), r->mrq(m+1, y));
    }
} *root;
```

1.4.6 Order Statistic Tree - More than K + Updates

```
struct node {
    ll s, e, m, it;
    ordered_set v;
    node *l, *r;
    node(ll _s, ll _e){
        s = _s; e = _e; m = (s+e)/2;
        if(s != e){
            l = new node(s, m);
            r = new node(m+1, e);
        }
```

```
    }
    void pu(ll x, ll y){
        if(s == e){ v.insert(pi(y, x)); return; }
        if(x > m) r->pu(x, y);
        if(x <= m) l->pu(x, y);
        v.insert(pi(y, x));
    }
    void pc(ll x, ll o, ll y){
        if(s == e){ v.clear(); v.insert(pi(y, x)); return; }
        if(x > m) r-> pc(x, o, y);
        if(x <= m) l->pc(x, o, y);
        v.erase(pi(o, x)); v.insert(pi(y, x));
    }
    ll rq(ll x, ll y, ll k){
        if(x > y) return 0;
        if(s == x && e == y){
            int ans = v.order_of_key(pi(k, LLONG_MAX));
            return v.size()-ans;
        }
        if(x > m) return r->rq(x, y, k);
        if(y <= m) return l->rq(x, y, k);
        return l->rq(x, m, k)+r->rq(m+1, y, k);
    }
} *root;
```

1.4.7 2D

```
struct node2D {
    int s, e, m;
    node1D *maxi;
    node2D *l, *r;
    node2D(int a, int b, int c, int d){
        s = a; e = b; m = (s+e)/2;
        maxi = new node1D(c, d);
        if(s != e){
            l = new node2D(s, m, c, d);
            r = new node2D(m+1, e, c, d);
        }
    }
    void pu(int a, int b, int v){
        if(s == e){ maxi->pu(b, v); return; }
        if(a <= m) l->pu(a, b, v);
        else r->pu(a, b, v);
        maxi->pu(b, max(l->maxi->rq(b, b), r->maxi->rq(b, b)));
    }
    int rq(int a, int b, int c, int d){
        if(s == a && e == b) return maxi->rq(c, d);
        if(b <= m) return l->rq(a, b, c, d);
        if(a > m) return r->rq(a, b, c, d);
        return max(l->rq(a, m, c, d), r->rq(m+1, b, c, d));
    }
} *root;
```

2 Graph Theory

Graph Algorithm	Complexity	Notes
DFS	O(N)	Flood Fill, Trees
BFS	O(N)	Unweighted Shortest Path
Floyd-Warshall	O(N^3)	All Pairs Shortest Path
Dijkstra	O(E log E)	Single Source Shortest Path
TSP	O(2^N)	Tour All Nodes
UFDS	O(1)	Checking Connectedness
MST - Kruskal	O(E log E)	Greedy
MST - Prim's	O(E log E)	Dijkstra
Bipartite Matching	O($N^{5/2}$)	2 Sets of Nodes
Articulation Points	O(N+E)	Find Splitting Nodes
Bridges	O(N+E)	Find Splitting Edges
SCC	O(N+E)	Nodes Can Reach Everyone

Table 2: Quick Summary of General Graph Algorithms

2.1 Depth First Search

Depth First Search goes as far in as possible, before coming all the way back out. It can be problematic on general graphs, but works fine and is easier to use for trees.
Time Complexity: O(N)

```
void dfs(int x, int p){
    for(auto it : adj[x]){
        if(it != p){
            par[it] = x;
            dep[it] = dep[x]+1;
            dist[it] = dist[x]+1;
            dfs(it, x);
        }
    }
}
```

2.2 Breadth First Search

Breadth First Search slowly spreads out from the source before going deeper in. It works well on general graphs but is slightly more tedious to code.
It can also find shortest paths on 0-1 weighted graphs. This is done using a deque and pushing 0s to the front and 1s to the back.
Time Complexity: O(N)

```
d[nx][ny] = 0;
q.push(pi(sx, sy));
while(!q.empty()){
    pi f = q.front(); q.pop();
    for(int i = 0; i < 4; i++){
        nx = f.first+dx[i];
        ny = f.second+dy[i];
        if(nx < 0 || ny < 0 || nx >= h || ny >= w) continue;
        if(d[nx][ny] != -1) continue;
        d[nx][ny] = d[f.first][f.second]+1;
        q.push(pi(nx, ny));
    }
}
```

2.3 Floyd-Warshall

Floyd-Warshall is very slow, so it is unlikely to be used. However, if there are too many edges, using Dijkstra from each node will still end up slower.
Time Complexity: O(N^3)

```
for(int i = 0; i < n; i++){
    for(int j = 0; j < n; j++){
        if(i == j) adj[i][j] = 0;
        else adj[i][j] = INT_MAX;
    }
}
for(int k = 0; k < n; k++){
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            adj[i][j] = min(adj[i][j], adj[i][k]+adj[k][j]);
        }
    }
}
```

2.4 Dijkstra

Dijkstra is much faster at shortest paths, but complexity is dependent on the number of edges.
Time Complexity: O(N)

```
priority_queue<pi, vector<pi>, greater<pi> > pq;
dist[s] = 0;
pq.push(pi(0, s));
while(!pq.empty()){
    pi f = pq.top(); pq.pop();
    for(auto it:adj[f.second]){
        if(dist[it.first] == -1 || dist[it.first] > f.first+it.second){
            dist[it.first] = f.first+it.second;
            pq.push(pi(dist[it.first], it.first));
        }
    }
}
```

2.5 Travelling Salesman Problem

The Travelling Salesman Problem uses DP on Bitmask to solve it. The Bitmask indicates which nodes have already been visited.
Time Complexity: O(2^N)

```
long long adj[14][14], dp[8200][14];

//start with 1, 0
//mask is 0 visited, at node 0
long long tsp(long long mask, long long pos){
    if(mask == visited) return adj[pos][0];
    if(dp[mask][pos] != -1) return dp[mask][pos];
    long long ans = LLONG_MAX;
    for(int i = 0; i < m; i++){
        if((mask&(1<<i)) == 0){
            long long newans = adj[pos][i]+tsp(mask|(1<<i), i);
            ans = min(ans, newans);
```

```
        }
    }
    return dp[mask][pos] = ans;
}


visited = (1 << m)-1;
res = tsp(1, 0);
if(res < 0) cout << "-1";
else cout << res;
```

2.6 Travelling Salesman Problem - BFS

```
Not Implemented Yet!
```

2.7 Union Find Disjoint Subset

Union Find Disjoint Subset connects subsets of nodes together. Using techniques like path compression reduce it's time complexity to O(1).
Time Complexity: O(1)

```
int root(int x){
    if(p[x] == -1) return x;
    return p[x] = root(p[x]);
}
void connect(int x, int y){
    p[root(x)] = root(y);
}
```

2.8 Minimum Spanning Tree

By stripping off larger edges, we are left with a tree such that the weights of edges between nodes are the minimum possible.
Time Complexity: O(E log E)

2.8.1 Kruskal

Kruskal is a greedy algorithm by processing starting from the smallest edge. It runs in O(E), but is O(E log E) because it needs to sort the edges.
Time Complexity: O(E log E)

```
sort(edgs.begin(), edgs.end());
for(auto it:edgs){
    if(root(it.second.first) != root(it.second.second)){
        connect(it.second.first, it.second.second);
        cost += it.first;
    }
}
```

2.8.2 Prim's

Prim's is literally Dijkstra, but using max instead of +. It can also be used to find minimum maximum edge from a single source.
Time Complexity: O(E log E)

```
priority_queue<pi, vector<pi>, greater<pi> > pq;
dist[s] = 0;
pq.push(pi(0, s));
while(!pq.empty()){
    pi f = pq.top(); pq.pop();
    for(auto it:adj[f.second]){
        if(dist[it.first] == -1 || dist[it.first] > max(f.first,it.second)){
            dist[it.first] = max(f.first, it.second);
            pq.push(pi(dist[it.first], it.first));
        }
    }
}
```

2.9 Topological Sort

A Topological Ordering is an ordering such that all nodes cannot be reached from any nodes after them. This can only be done on Directed Acyclic Graphs (DAGs).
Time Complexity: O(N log N)

```
void dfs(int id){
    if(visited[id]) return;
    visited[id] = 1;
    for(auto it : adj[id]) dfs(it);
    ranking.push_back(id);
}
```

2.10 Bipartite Matching

A Bipartite Graph has nodes in 2 sets, such that edges only run across both sets. A chooses edges such that each node only has a maximum of one edge.
Time Complexity: O($N^{5/2}$)

```
bool dfs(int u){
    if(v[u]) return 0;
    v[u] = 1;
    for(auto v:adj[u]){
        if(match[v] == -1 || dfs(match[v])){
            match[v] = u;
            match[u] = v;
            return 1;
        }
    }
    return 0;
}
memset(match, -1, sizeof(match));
for(auto lit:ho){
    for(auto rit:adj[lit]){
        if(match[rit] == -1){
            match[rit] = lit;
            match[lit] = rit;
            mcbm++;
            break;
        }
    }
}
for(auto lit:ho){
```

```
    if(match[lit] == -1){
        memset(v, 0, sizeof(v));
        mcbm += dfs(lit);
    }
}
mis = n-mcbm;
```

2.11 Articulation Points

Articulation Points are nodes that split the graph when they are removed. They are found using Tarjan's.
Time Complexity: O(N+E)

```
void atp(int node, int depth){
    vi[node] = 1;
    dep[node] = depth;
    low[node] = depth;
    for(auto it:adj[node]){
        if(!vi[it]){
            par[it] = node;
            atp(it, depth+1);
            chi[node]++;
            if(low[it] >= dep[node]) atps[node]++;
            low[node] = min(low[node], low[it]);
        } else if(it != par[node]){
            low[node] = min(low[node], dep[it]);
        }
    }
}
atp(0, 0);
// root -> chi[i]
// other -> atps[i]+1
```

2.12 Bridges

Bridges are edges that split the graph when they are removed. They are found using Tarjan's.
Time Complexity: O(N+E)

```
void bridges(int x, int par){
    if(dep[x] != -1) return;
    dep[x] = low[x] = co++;
    int t = 0;
    for(auto it:adj[x]){
        if(it == par && t == 0){ t++; continue; }
        if(dep[it] != -1){
            if(low[it] > dep[x]) bgs.push_back(pi(x, it));
            low[x] = min(low[x], low[it]);
            continue;
        }
        bridges(it, x);
        if(low[it] > dep[x]) bgs.push_back(pi(x, it));
        low[x] = min(low[x], low[it]);
    }
}
for(int i = 1; i <= n; i++) bridges(i, 0);
```

2.13 Strongly Connected Components

Strongly Connected Components are sets of nodes where all nodes can visit each other. After compressing them into single nodes, the new graph formed is a DAG (Directed Acyclic Graph).
Time Complexity: O(N+E)

```
stack<int> s;
vector<int> cur, adj[100001];
set<int> adjscc[100001];
vector<vector<int> > comps;
void scc(int v){
    idxs[v] = idx;
    lowlink[v] = idx;
    idx++;
    s.push(v);
    ins[v] = 1;
    for(auto w:adj[v]){
        if(idxs[w] == -1){
            scc(w);
            lowlink[v] = min(lowlink[v], lowlink[w]);
        } else if(ins[w]){
            lowlink[v] = min(lowlink[v], idxs[w]);
        }
    }
    if(lowlink[v] == idxs[v]){
        cur.clear();
        w = 0;
        while(w != v){
            w = s.top(); s.pop();
            ins[w] = 0;
            cur.push_back(w);
        }
        comps.push_back(cur);
    }
}
idx = 1;
memset(idxs, -1, sizeof(idxs));
for(int i = 1; i <= n; i++){
    if(idxs[i] == -1) scc(i);
}
memset(com, -1, sizeof(com));
for(int i = 0; i < comps.size(); i++){
    for(auto it:comps[i]) com[it] = i;
}
for(int i = 1; i <= n; i++){
    for(auto it:adj[i]){
        if(com[i] != com[it]) adjscc[com[i]].insert(com[it]);
    }
}
```

2.14 Trees

2.14.1 Diameter

The diameter of a tree is the longest possible distance between 2 nodes. The method of finding it is finding the farthest node from a source, then finding the farthest node from this new source.
The centroid of a tree (node that minimises sum of distances) is the node on the diameter where the distance is closest to the length of the diameter/2.
Time Complexity: O(N)

```
pi dfs(int node, int par, int dist){
    pi b = pi(node, dist);
    for(auto it:adj[node]){
        if(it.first != par){
            pi c = dfs(it.first, node,dist+it.second);
            if(c.second > b.second) b = c;
        }
    }
    return b;
}
f = dfs(0, -1, 0);
l = dfs(f.first, -1, 0);
// l.second
```

2.14.2 2^K Decomposition

By using Precomputation, we can find the Kth parent of any node quickly.
Time Complexity: O(N log N)

```
int par(int x, int k){
    for(int i = 19; i >= 0; i--){
        if(k >= (1<<i)){
            if(x == -1) return x;
            x = p[x][i];
            k -= (1<<i);
        }
    }
    return x;
}

memset(p, -1, sizeof(p));
dfs(0);
for(int k = 1; k <= 19; k++){
    for(int i = 0; i < n; i++){
        if(p[i][k-1] != -1) p[i][k] = p[p[i][k-1]][k-1];
    }
}
```

2.14.3 Lowest Common Ancestor

When performing calculations on trees, a path between 2 nodes must run between them and a lowest common ancestor. This is also needed as Precomputation using a DFS downwards only gives distance from the root.
Time Complexity: O(N log N)???

```
int lca(int x, int y){
    if(dep[x] < dep[y]) swap(x, y);
    for(int k = 19; k >= 0; k--){
        if(p[x][k] != -1 && dep[p[x][k]] >= dep[y]) x = p[x][k];
    }
    if(x == y) return x;
    for(int k = 19; k >= 0; k--){
        if(p[x][k] != p[y][k]){
            x = p[x][k];
            y = p[y][k];
        }
```

```
    }
    return p[x][0];
}
```

2.14.4 All Pairs Shortest Path

Finding the distance between any pair of nodes in a tree can be done very quickly. A DFS from the root can be performed giving us distances from it to our 2 nodes. This leads to unecessary distance between the root and LCA being overcounted twice.
Time Complexity: O(1)

```
int distance(int x, int y){
    return dist[x]+dist[y]-2*dist[lca(x, y)];
}
```

We sometimes want to relabel a tree's nodes to give them a new order, maybe to make subtrees easier to label. This is where Preorder and Postorder come in.
Time Complexity: O(N)

2.14.5 Preorder

```
// UNTESTED
void dfs(int x, int par){
    c++;
    pre[x] = c;
    for(auto it:adj[x]){
        if(it != par) dfs(adj[it], x);
    }
}
```

2.14.6 Postorder

```
// UNTESTED
void dfs(int x, int par){
    c++;
    for(auto it:adj[x]){
        if(it != par) dfs(adj[it], x);
    }
    post[x] = c;
}
```

2.14.7 Subtree to Range

Sometimes we may need to perform range updates on a subtree, such as when updating distances. This allows a subtree to be represented as a continuous subarray in a Data Structure.
Time Complexity: O(N)

```
int dfs(int x, int par){
    c++;
    pre[x] = c;
    for(auto it:adj[x]){
        if(it != par) rig[pre[x]] = max(rig[pre[x]], dfs(it, x));
    }
    if(rig[pre[x]] == 0) rig[pre[x]] = pre[x];
    return rig[pre[x]];
```

```
}
// node -> pre[x]
// children -> pre[x]+1, rig[pre[x]]
```

2.14.8 Leaf Pruning

In some cases, we only need to consider certain important nodes. To make calculations easier we can first remove extra leaf nodes.
Time Complexity: O(N log N)

```
for(int i = 1; i <= n; i++){
    if(adj[i].size() == 1) leafs.push(i);
}
while(!leafs.empty()){
    u = leafs.front(); leafs.pop();
    t = adj[u][0];
    adj[t].erase(find(adj[t].begin(), adj[t].end(), u));
    if(adj[t].size() == 1) leafs.push(t);
    adj[u].erase(find(adj[u].begin(), adj[u].end(), t));
}
```

2.14.9 Weighted Maximum Independent Set

This may sound similar to MIS on a Bipartite graph, where no 2 nodes can be adjacent. A naive DFS with alternation doesn't work as it may be optimal to have longer stretches where we do not take nodes.
Time Complexity: O(N)

```
int mis(int v, bool take, int p){
    if(dp[v][take] != -1) return dp[v][take];
    int ans = take*c[v];
    for(auto it:adj[v]){
        if(it == p) continue;
        int temp = mis(it, 0, v);
        if(!take) temp = max(temp, mis(it, 1, v));
        ans += temp;
    }
    return dp[v][take] = ans;
}
void ans(int v, bool take, int p){
    for(auto it:adj[v]){
        if(it == p) continue;
        int temp0 = dp[it][0], temp1 = (take ? -1 : dp[it][1]);
        if(temp0 > temp1) ans(it, 0, v);
        else { a.push_back(it); ans(it, 1, v); }
    }
}
mis(0, 0, -1);
mis(0, 1, -1);
if(dp[0][1] > dp[0][0]){ a.push_back(0); ans(0, 1, -1); }
else ans(0, 0, -1);
```

2.14.10 Heavy-Light Decomposition

This is similar to the principle of converting Subtrees to Ranges, but we cannot always update a whole subtree. We decompose the whole tree into chains, which can be rearranged to form a chain of chains in a Data Structure.

Time Complexity: O(N)
Query: O(N log N) (Accounting for log N DS)

```
int dfs(int node){
    int size = 1, max_c = 0;
    for(auto it:adj[node]){
        if(it.s != par[node]){
            par[it.s] = node; dep[it.s] = dep[node]+1;
            int c_size = dfs(it.s);
            size += c_size;
            if(c_size > max_c){ max_c = c_size; heav[node] = it.s; }
        }
    }
    return size;
}
void decomp(int node, int hea){
    head[node] = hea; pos[node] = c_pos++;
    if(heav[node] != -1) decomp(heav[node], hea);
    for(auto it:adj[node]){
        if(it.s == par[node]) continue;
        if(it.s != heav[node]) decomp(it.s, it.s);
        root->update(pos[it.s], it.f);
    }
}
int query(int a, int b){
    int res = 0;
    for(; head[a] != head[b]; b = par[head[b]]){
        if(dep[head[a]] > dep[head[b]]) swap(a, b);
        res = max(res, root->query(pos[head[b]], pos[b]));
    }
    if(dep[a] > dep[b]) swap(a, b);
    res = max(res, root->query(pos[a]+1, pos[b]));
    return res;
}
dfs(0);
decomp(0, 0);
```

2.14.11 Centroid Decomoposition

We use this to calculate the minimum distance from a node to a set of nodes. We decompose the tree into centroids where have at most N/2 children in each of it's subtrees. This reduces the overall height of the tree and generates a new centroid tree.
Time Complexity: O(N)
Update: O(log N)???
Query: O(log N)???

```
ll dfs1(ll u, ll p, ll l){
    sub[u] = 1;
    for(auto it : adj[u]){
        if(ban[it.f] != -1) continue;
        if(it.f == p) continue;
        if(l) dst[it.f][l-1] = dst[u][l-1]+it.s;
        sub[u] += dfs1(it.f, u, l);
    }
    return sub[u];
}

```

```
ll dfs2(ll u, ll p, ll n){
    for(auto it : adj[u]){
        if(ban[it.f] != -1) continue;
        if(it.f != p && sub[it.f] > n/2){
            return dfs2(it.f, u, n);
        }
    }
    return u;
}

void build(ll u, ll p, ll l){
    ll n = dfs1(u, p, l);
    ll cent = dfs2(u, p, n);
    if(p == -1) p = cent;
    par[cent] = p;
    ban[cent] = l;
    for(auto it : adj[cent]){
        if(ban[it.f] != -1) continue;
        dst[it.f][l] = it.s;
        build(it.f, cent, l+1);
    }
}

void update(ll x){
    ll lvl = ban[x];
    ll y = x;
    while(lvl != -1){
        ans[y] = min(ans[y], dst[x][lvl]);
        st.push(y);
        y = par[y];
        lvl--;
    }
}

ll query(ll x){
    ll res = LLONG_MAX/3;
    ll lvl = ban[x];
    ll y = x;
    while(lvl != -1){
        res = min(res, ans[y]+dst[x][lvl]);
        y = par[y];
        lvl--;
    }
    return res;
}

//Initialisation
memset(ban, -1, sizeof(ban));
build(0, -1, 0);
for(ll i = 0; i < n; i++) ans[i] = LLONG_MAX/3;

//Adding a node to be considered
update(X);

//Querying with all considered nodes
query(Y);
while(!st.empty()){ ans[st.top()] = LLONG_MAX/3; st.pop(); }
```

3 Dynamic Programming

DP Algorithm	Complexity	Notes
Coin Change	O(NV)	
Coin Combinations	O(NV)	
Knapsack - 0-1	O(NS)	
Knapsack - 0-K	O(log2(K)+NS)	
LIS - Naive	O(N^2)	
LIS - DS / Optimal	O(N log N)	
LCS	O(N^2)	
LCS - LIS	O(N log N)	
Digits	O(10—N—)	
Convex Hull Speedup	O(N)	Amortized
Divide and Conquer	O(N log N)	

Table 3: Quick Summary of General Dynamic Programming Algorithms

3.1 Coin Change

```
dp[0] = 0;
for(int i = 1; i <= v; i++){
    dp[i] = INT_MAX;
    for(int j = 0; j < n; j++){
        if(i >= c[j]) dp[i] = min(dp[i], dp[i-c[j]]+1);
    }
}
```

3.2 Coin Combinations

```
ways[0] = 1;
for(int i = 0; i < c; i++){
    for(int j = 1; j <= v; j++){
        if(j >= coins[i]) ways[j] += ways[j-coins[i]];
    }
}
```

3.3 Knapsack

3.3.1 0-1

```
for(int i = 0; i < n; i++){
    for(int j = s; j >= w[i]; j--){
        dp[j] = max(dp[j], dp[j-w[i]]+v[i]);
    }
}
```

3.3.2 0-K

```
//further speedup:
//take top s/w most valued items
//for every possible item
//not just multiple copies of one item

for(int i = 0; i < n; i++){
```

```
    cin >> v >> w >> k;
    k = min(k, s/w);
    c = 1;
    while(true){
        if(k < c) break;
        items.push_back(pi(v*c, w*c));
        k -= c;
        c *= 2;
    }
    if(k != 0) items.push_back(pi(v*k, w*k));
}
for(auto it :items){
    for(int i = s; i >= it.second; i--){
        dp[i] = max(dp[i], dp[i-it.second]+it.first);
    }
}
```

3.4 Longest Increasing Subsequence

3.4.1 N^2

```
for(int i = 0; i < n; i++){
    for(int j = 0; j < i; j++){
        if(a[j] < a[i]) lis[i] = max(lis[i], lis[j]);
    }
    lis[i]++;
    ans = max(ans, lis[i]);
}
```

3.4.2 $NlogN$

```
for(int i = 0; i < n; i++){
    t = query(a[i]-1)+1;
    update(a[i], t);
    ans = max(ans, t);
}
```

3.4.3 Optimal

```
for(int i = 0; i < n; i++){
    int p = lower_bound(dp, dp+l, a[i])-dp;
    dp[p] = a[i];
    l = max(p+1, l);
}
```

3.5 Longest Common Subsequence

3.5.1 N^2

```
int lcs(int al, int bl){
    if(al == 0 || bl == 0) return 0;
    if(dp[al][bl] != -1) return dp[al][bl];
    dp[al][bl] = max(lcs(al-1, bl), lcs(al, bl-1));
    if(a[al-1] == b[bl-1]) dp[al][bl] = max(dp[al][bl], lcs(al-1, bl-1)+1);
    return dp[al][bl];
```

```
}
```

3.5.2 Longest Increasing Subsequence

```
for(int i = 0; i < n; i++){ cin >> a[i].first; a[i].second = i+1; }
for(int i = 0; i < m; i++) cin >> b[i];
sort(a, a+n);
for(int i = 0; i < m; i++){
    p = lower_bound(a, a+n, pi(b[i], 0));
    if(p != a+n) c.push_back(p->second);
}
// Perform LIS on c
```

3.6 Digits

```
long long derp(int idx, int prev, int same, int allzero){
    if(idx == num.size()) return dp[idx][prev][same][allzero] = !allzero;
    if(dp[idx][prev][same][allzero] != -1) return dp[idx][prev][same][allzero];
    long long sum = 0, limit;
    if((!allzero && same) || (allzero && idx == 0)) limit = num[idx];
    else limit = 9;
    for(int i = 0; i <= limit; i++){
        if(i == 4) continue;
        if(allzero){
            if(i == 0) sum += derp(idx+1, 10, 1, 1);
            else sum += derp(idx+1, i, (idx == 0 && i == limit), 0);
        } else if(!(i == 3 && prev == 1)){
            if(same && i == num[idx]) sum += derp(idx+1, i, 1, 0);
            else sum += derp(idx+1, i, 0, 0);
        }
    }
    return dp[idx][prev][same][allzero] = sum;
}
void dcmp(long long x){
    num.clear();
    while(x > 0){ num.push_back(x%10); x /= 10; }
    reverse(num.begin(), num.end());
}
long long solve(long long x){
    if(memo.count(x) != 0) return memo[x];
    memset(dp, -1, sizeof(dp));
    dcmp(x);
    return memo[x] = derp(0, 10, 1, 1);
}
```

3.7 Convex Hull Speedup

This speeds up any DP which is a quadratic function. A similar idea can also be done for linear functions but just using a set.
The important part is knowing how to rearrange the transition to get coefficients.
Query: O(1) Update: O(1)

```
// dp(x) = max(dp(i)+f(p(x)-p(i)))
//       = max(dp(i)+a(p(x)-p(i))^2+b(p(x)-p(i))+c)
//       = max(dp(i)+ap(x)^2-2ap(x)p(i)+ap(i)^2+bp(x)-bp(i))+c
```

```
//        = max(dp(i)+ap(i)^2-bp(i)-2ap(x)p(i))+ap(x)^2+bp(x)+c
//        = max(c(i)+m(i)p(x))+v(x)
// c(i) = dp(i)+ap(i)^2-bp(i)
// m(i) = -2ap(i)
// v(x) = ap(x)^2+bp(x)+c
long long func(pi line, long long x){
    return line.first*x+line.second;
}
long double intersection(long long m1, long long c1, long long m2, long long c2){
    return (long double)(c2-c1)/(m1-m2);
}
long double intersect(pi x, pi y){
    return intersection(x.first, x.second, y.first, y.second);
}
long long query(long long x){
    while(hull.size() > 1){
        if(func(hull[0], x) < func(hull[1], x)){
            hull.pop_front();
        } else break;
    }
    return func(hull[0], x);
}
void insert(long long m, long long c){
    pi line = pi(m, c);
    while(hull.size() > 1){
        long long s = hull.size();
        if(intersect(hull[s-1], line) <= intersect(hull[s-2], line)){
            hull.pop_back();
        } else break;
    }
    hull.push_back(line);
}
insert(0, 0);
for(int i = 1; i <= n; i++){
    dp[i] = query(ps[i])+a*ps[i]*ps[i]+b*ps[i]+c;
    insert(-2*a*ps[i], dp[i]+a*ps[i]*ps[i]-b*ps[i]);
}
```

3.8 Divide and Conquer

```
long long cost(int s, int e){
    return (ps[e]-ps[s-1])*(e-s+1);
}
void dnc(int s, int e, long long x, int y, int k){
    if(s > e) return;
    int m = (s+e)/2, best = 0;
    dp[m][k] = LLONG_MAX/2;
    for(int i = x; (i <= y && i <= m); i++){
        if(dp[m][k] > dp[i][!k]+cost(i+1, m)){
            dp[m][k] = dp[i][!k]+cost(i+1, m);
            best = i;
        }
    }
    if(s < m) dnc(s, m-1, x, best, k);
    if(m < e) dnc(m+1, e, best, y, k);
}
for(int i = 1; i <= n; i++) dp[i][0] = LLONG_MAX/2;
```

```
for(int i = 1; i <= g; i++){
    for(int j = 1; j <= n; j++) dp[j][i%2] = LLONG_MAX/2;
    dnc(0, n, 0, n, i%2);
}
```

4 Math

4.1 Greatest Common Divisor

```
// Alternatively, find highest common powers for each factor
int gcd(int a, int b){
    if(b == 0) return a;
    return gcd(b, a%b);
}
```

4.2 Lowest Common Multiple

```
// Alternatively, find highest powers for each factor
int lcm(int a, int b){
    return (a*b)/gcd(a, b);
}
```

4.3 Modular Functions

4.3.1 Multiplication

```
int mulmod(int a, int b, int m) {
    int res = 0;
    while(b > 0) {
    if(b % 2 == 1) res = (res+a)%m;
        a = (a*2)%m;
        b /= 2;
    }
    return res % m;
}
```

4.3.2 Exponentiation

```
int powmod(int a, int b, int m) {
    int res = 1;
    while(b > 0) {
    if(b % 2 == 1) res = mulmod(res, a, m);
        a = mulmod(a, a, m);
        b /= 2;
    }
    return res % m;
}
```

4.3.3 Inverse

```
ll modinv(ll a){
    return powmod(a, MOD-2, MOD);
}
```

4.4 Primes

4.4.1 Sieve of Eratosthenes

```
memset(prime, 1, sizeof(prime));
prime[0] = prime[1] = 0;
for(int i = 2; i <= 1000000; i++){
    if(prime[i]){
        for(int j = 2; i*j <= 1000000; j++){
            prime[i*j] = 0;
        }
    }
}
```

4.4.2 Prime Factorisation

```
while(x % 2 == 0){ cnt[2]++; x /= 2; }
for(int i = 3; i*i <= x+1; i += 2){
    while(x % i == 0){ cnt[i]++; x /= i; }
    if(x == 1) break;
}
if(x > 1) cnt[x]++;
```

4.5 Fibonacci

```
typedef pair<pi, pi> matrix;
matrix multiply(matrix x, matrix y, long long z){
    return matrix(
            pi((x.f.f*y.f.f+x.f.s*y.s.f)%z,
                (x.f.f*y.f.s+x.f.s*y.s.s)%z),
            pi((x.s.f*y.f.f+x.s.s*y.s.f)%z,
                (x.s.f*y.f.s+x.s.s*y.s.s)%z));
}
matrix square(matrix x, long long y){
    return multiply(x, x, y);
}
matrix power(matrix x, long long y, long long z){
    if(y == 1) return x;
    if(y % 2 == 0) return square(power(x, y/2, z), z);
    return multiply(x, square(power(x, y/2, z), z), z);
}
long long fibo(long long n, long long m){
    matrix x = matrix(pi(1, 1), pi(1, 0));
    return power(x, n, m).f.s;
}
```

4.6 ${}^nC_k$

```
//Naive
if(k > n-k) k = n-k;
for(int i = 0; i < k; i++){
    ans *= (n-i);
    ans /= (i+1);
}

```

```
//Precomputed
//fac -> factorial
//mi -> modular inverse
ll nck(ll n, ll k){
    if(n < k) return 0;
    return (((fac[n]*mi[k])%MOD)*mi[n-k])%MOD;
}
```

4.7 Expected Value

Expected Value is the expected result when an event happens once. e.g.
Expected Value = Sum of Possible Values / Number of Possible Values
Expected Distance = Sum of All Possible Distances / Number of Paths

5 Algorithms

5.1 Discretisation

```
for(int i = 0; i < n; i++){
    cin >> a[i];
    b[i] = a[i];
}
sort(b, b+n);
for(int i = 0; i < n; i++){
    d = lower_bound(b, b+n, a[i])-b;
    a[i] = d+1;
}
```

5.2 Binary Search

```
while(mini < maxi){
    medi = mini+(maxi-mini)/2;
    if(can(medi)) maxi = medi;
    else mini = medi+1;
}
```

5.3 Meet in the Middle

```
//Example - Bobek
//Count number of subsets with sum <= m
o = n/2; t = n-o;
//Generate first half of subsets
for(int i = 0; i < (1 << o); i++){
    c = bitset<50>(i);
    for(int j = 0; j < o; j++){
        if(c.test(j)) s[i] += a[j];
    }
}
sort(s, s+(1 << o));
//Generate second half of subsets
for(int i = 0; i < (1 << t); i++){
    c = bitset<50>(i); sum = 0;
    for(int j = 0; j < t; j++){
        if(c.test(j)) sum += a[o+j];
```

```
    }
    ans += upper_bound(s, s+(1<<o), m-sum)-s;
}
```

5.4 Mo's Algorithm

```
//queries : index, left, right
//answers : index, answer

bool cmp(pii x, pii y){
    if(x.second.first/blk != y.second.first/blk){
        return x.second.first/blk < y.second.first/blk;
    }
    if(x.second.first/blk & 1){
        return x.second.second < y.second.second;
    }
    return x.second.second > y.second.second;
}

blk = sqrt(n);
sort(qs, qs+q, cmp);
for(int i = 0; i < q; i++){
    l = qs[i].second.first; r = qs[i].second.second;
    while(lft > l){
        //remove a[lft-1]
        lft--;
    }
    while(rgt <= r){
        //add a[rgt]
        rgt++;
    }
    while(lft < l){
        //add a[lft]
        lft++;
    }
    while(rgt > r+1){
        //remove a[rgt-1]
        rgt--;
    }
    ans[i] = pi(qs[i].first, cur);
}
sort(ans, ans+q);
```

5.5 Sliding Set

Sometimes our DP state involves taking the minimum from some previously calculated numbers. Instead of using a for loop, using a data structure like a set will speed this up.
Query: O(1) Update: O(log N) Total Updates: O(N log N)

```
//Example - Candymountain
for(int i = 0; i < k; i++) s.insert(dp[i]);
for(int i = k; i < n; i++){
    dp[i] = max(dp[i], *s.begin());
    s.erase(s.find(dp[i-k]));
    s.insert(dp[i]);
}
```

5.6 Sliding Deque

The sliding set isn't the fastest we can speed it up using data structures. In this case, maintain a deque of strictly increasing elements.
Adding a number: Pop the back until it is less than the number to add. Push that number to the back. Removing a number: Check if the front number is the number to remove. If it is, remove it. Querying: Since the front is the minimum possible and the maximum is our current element, the answer is the difference between these.
Query: O(1) Total Updates: O(N)

```
//Example - Supermarket
for(int i = 1; i <= n; i++){
    curw += w[i];
    //Remove the back until it is lower
    while(dq.size() > 0 && dq.back() > pd[i-1]) dq.pop_back();
    //Push current element
    dq.push_back(pd[i-1]);
    //Remove invalid items from the front
    while(curs < i && curw >= x){
        if(dq.front() == pd[curs-1]) dq.pop_front();
        curw -= w[curs];
        curs++;
    }
    //Take maximum range of front to back
    ans = max(ans, pd[i]-dq.front());
}

//Example - Rubies
for(int i = 1; i <= n; i++)
    ps[i] = ps[i-1]+c[i]-x*w[i];
    //Remove invalid items from the front
    while(dq.size() > 0 && (i-dq.front().S+1) > h) dq.pop_front();
    if(wait.size() > 0 && ((i-wait.front().S+1) >= l)){
        //Remove the back until it is lower
        while(dq.size() > 0 && dq.back().F > wait.front().F) dq.pop_back();
        //Push current element
        dq.push_back(wait.front()); wait.pop_front();
    }
    //Take maximum range of front to back
    if(dq.size() > 0 && ((i-dq.front().S+1) >= l) && ps[i] >= dq.front().F) return
    wait.push_back(pi(ps[i-1], i));
    if(l == 1 && ps[i] >= ps[i-1]) return true;
}
```

6 Miscellaneous

6.1 Macros + Functions + Variables

```
#pragma GCC optimize("O3")
#pragma GCC optimize("unroll-loops")

#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;
using namespace std;
```

```

#define EPS 1e-9
#define INF LLONG_MAX/3LL
#define MOD 1e9+7
#define F first
#define S second
#define pf push_front
#define pb push_back
#define pof pop_front
#define pob pop_back
#define ins insert
#define lb lower_bound
#define ub upper_bound
#define sz(a) ((int)(a).size())
#define all(a) begin(a), end(a)
#define FOR(i, a, b) for(int i = a; i <= b; i++)
#define ROF(i, a, b) for(int i = a; i >= b; i--)
#define F0R(i, a) FOR(i, 0, a)
#define R0F(i, a) ROF(i, a, 0)
#define ITER(i, a) for(auto i : a)
#define FAST ios_base::sync_with_stdio(false); cin.tie(0); cout.tie(0);

typedef long long ll;
typedef pair<ll, ll> pi;
typedef pair<ll, pi> pii;
typedef tree<pi, null_type, less<pi>,
             rb_tree_tag, tree_order_statistics_node_update> ordered_set;

int ls(int x){ return (x)&(-x); }
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
inline ll rngr(ll x, ll y) { return (rng()%(y-x+1))+x; }
inline ll ri() {
    ll x = 0;
    char ch = getchar_unlocked();
    while (ch < '0' || ch > '9') ch = getchar_unlocked();
    while (ch >= '0' && ch <= '9'){
        x = (x << 3) + (x << 1) + ch - '0';
        ch = getchar_unlocked();
    }
    return x;
}

// 4 Directions
int dx[]={0, 0, -1, 1};
int dy[]={-1, 1, 0, 0};
// 8 Directions
int dx[]={0, 0, -1, 1, -1, 1, -1, 1};
int dy[]={-1, 1, -1, 1, 0, 0, 1, -1};
// Knight Moves
int dx[]={-1, -2, 1, 2, 2, 1, -2, 1};
int dy[]={-2, -1, -2, -1, 1, 2, 1, 2};
```

6.2 Compile Commands

```
g++ -c "%f" -std=c++11 //Editor Compile
g++ -o "%e" "%f" -std=c++11 //Editor Build

```

```
g++ "file.cpp" -o "file" -std=c++11 //Simple for command line
```

6.2.1 Simple Script

```
int main(int argc, char **argv){
    system(g++ -o (argv[1]) (argv[1]).cpp -std=c++11);
}
```

6.3 Pruning

```
auto start = chrono::high_resolution_clock::now();
auto end = chrono::high_resolution_clock::now();
auto elapse = chrono::duration<double>(end-start);
if(elapse.count() > 2.9) break;
```

6.4 Optimise

```
int __attribute__((optimize("Ofast"), target("arch=sandybridge"))) f(){}
```

7 Information

7.1 Time complexity v/s N

Complexity	Maximum in 1s
O(1)	Infinite
O(log N)	2^{10^6}
O(sqrt N)	10^{14}
O(N)	10^7
O(N log N)	10^6
O(N sqrt N)	10^5
O(N^2)	10^4
O(N^3)	500
O(N^4)	100
O(2^N)	22
O(N x 2^N)	20
O(N!)	12
O(N x N!)	11

Table 4: Quick Summary of Complexities and Maximum Input

7.2 STL Data Structures / Functions

```
//File I/O
freopen("test.in", "r", stdin);
freopen("test.out", "w", stdout);

//Variable / Array Functions
min(a, b);
max(a, b);
__gcd(a, b);
swap(a, b);

```

```
lower_bound(a, a+n, b); //Value >= b
upper_bound(a, a+n, b); //Value > b

fiil(a, a+n, 0);
memset(a, 0, sizeof(a));
copy(a, a+n, b);

sort(a, a+n);
stable_sort(a, a+n); //If a = b, indexes will remain same
reverse(a, a+n);
random_shuffle(a, a+n, rng);
max_element(a, a+n);

//Math Functions (Low Accuracy)
pow(1, 2);
sqrt(1);
cbrt(1);
floor(1);
ceil(1);
abs(1);
log(1);
log10(1);

//Limits
INT_MAX
LLONG_MAX
LDBL_MAX

//Others
to_string(0);
stoll("0");

vector<int> v;
v.push_back(0);
v.front(); //0
v.back(); //0
v.pop_back();

queue<int> q;
q.push(0);
q.front(); //0
q.pop();

priority_queue<int> pq;
pq.push(0)
pq.top(); //0
pq.pop();

deque<int> dq;
dq.push_front(0);
dq.front(); //0
dq.push_back(1);
dq.back(); //1
dq.pop_front();
dq.pop_back();

set<int> s;
s.insert(0);
```

```
69  s.find(0); // Pointer to 0's position
70  s.erase(0);
71
72  multiset<int> ms;
73  ms.insert(0);
74  ms.insert(0);
75  ms.count(0); //2
76  ms.find(0); //Pointer to a 0's position
77  ms.erase(ms.find(0)); //Erase a 0
78  ms.erase(0); //Erase all 0s
79
80  map<int, int> m;
81  m[0] = 1;
82  m[1] = 2;
83
84  stack<int> st;
85  st.push(0);
86  st.top(); //0
87  st.pop();
88
89  typedef tree<int, null_type, less<int>,
90              rb_tree_tag, tree_order_statistics_node_update> ordered_set;
91
92  typedef tree<pi, null_type, less<pi>, //Use a pair to simulate multisets
93              rb_tree_tag, tree_order_statistics_node_update> ordered_set;
94
95  ordered_set os;
96  os.insert(1);
97  os.insert(2);
98  os.insert(4);
99  os.insert(8);
100 os.insert(16);
101 //Find by order - Kth largest (starting from 0)
102 os.find_by_order(1)<<endl; // 2
103 os.find_by_order(2)<<endl; // 4
104 os.find_by_order(4)<<endl; // 16
105 //Order of key - How many elements are < K
106 os.order_of_key(-5)<<endl;  // 0
107 os.order_of_key(1)<<endl;   // 0
108 os.order_of_key(3)<<endl;   // 2
109 os.order_of_key(4)<<endl;   // 2
110 os.order_of_key(400)<<endl; // 5
```