CS345 : Assignment 1

Dhruv Gupta (220361)

August 2024

Dhruv Gupta: 220361

Question 1

Let the centre of the unit circle be the origin O. **Let us define polar angle of a point P as the angle between positive x-axis and line joining O and P, measured anti clockwise.**

We will first create two arrays A and B such that

$$A[i] = polar\ angle\ of\ p_i \tag{1}$$

$$B[i] = polar\ angle\ of\ q_i \tag{2}$$

Now we will modify the two arrays so that

$$A[i] = min(A[i], B[i]) \tag{3}$$

$$B[i] = max(A[i], B[i]). \tag{4}$$

Let us denote the line segment formed by joining p_i and q_i be L_i. Note that L_i is uniquely determined by its two end points and each end point is uniquely determined by its polar angle. **We will call a point P smaller than point Q if polar angle of P < polar angle of Q**. Note that $A[i]$ and $B[i]$ store the polar angles of the smaller and larger endpoints of L_i respectively.

Condition for two line segments (chords) to intersect :

Two chords L_i and L_j intersect if $A[i] < A[j] < B[i] < B[j]$ or $A[j] < A[i] < B[j] < B[i]$.

Description of Data Structure used :

We will create a augmented balanced BST. Each chord will be represented in the tree by a node. Each node N of the tree will store the following data - A, B, B_arr, $size$, $left$ and $right$. A and B are the polar angles of the chord ($A < B$). B_arr is a sorted array of size $size$ which stores the B values of all the nodes in the subtree rooted at the node N in increasing order. Nodes in the tree are compared by their A values, i.e., inorder traversal of the tree will list the nodes in their increasing order of A values. The $left$ and $right$ are pointers pointing to the left and right child of the node N.

Description of the algorithm :

- First we will create the balanced BST.
  - Make an array $Chords$ of n nodes. $Chords[i]$ will be a node containing the following - $A = A[i]$, $B = B[i]$, an empty array B_arr, $size = 0$, $left = right = NULL$.
  - Sort the array $Chords$ in increasing order of A values. Create a balanced BST T from this sorted array.
  - Apply DFS to fill the $size$ and B_arr variables. This can be easily done recursively. Let p be the parent node and q, r be its children. Then, $p.size = q.size + r.size + 1$ and $p.B\_arr = Merge(q.B\_arr, r.B\_arr)$.
- For each element c_i in $Chords$ array, we will find the number of elements c_j such that $j > i$ and c_i, c_j intersect, i.e., $c_i.A < c_j.A < c_i.B < c_j.B$. Note that since $i < j$ and the $Chords$ array is sorted in increasing order of A values, the other condition for intersection is not possible. Also, if we count this way, there will be no repetitions and all intersections are counted exactly once.
- Consider a particular element c. We will first find all the nodes with A value between $c.A$ and $c.B$. Then among all those nodes we will count the number of nodes with B value greater than $c.B$. This will give me the count of nodes which satisfy the intersection condition.
  - Consider two nodes in the tree T - node c_1 with A value $c.A$ and the node c_2 with A value $c.B$. If no such c_2 exists, take the node which is the predecessor of $c.B$, i.e., the node with the largest A value smaller than $c.B$.
  - Let the LCA of c_1 and c_2 be c_0. Consider the path P_L from c_0 to c_1 and the path P_R from c_0 to c_2. We will use nodes on this path to find the number of intersections.
  - Let us make a set S of all the nodes which we need to consider for counting intersections. We will also maintain a variable *count* to keep track of intersecting nodes we have found so far.
  - Start from the root of T. We want to search for c_0. Note that c_0 will be the node with the minimum depth (depth is the number of edges on the path from root) such that $c_1.A < c_0.A < c_2.A$, so if the current node is c' and $c'.A < c_1.A$, we move to its right child, and if $c'.A > c_2.A$, we move to its left child, otherwise, we have found c_0' and we stop. Once c_0 is found, check if $c_0.B > c.B$, if yes set *count* $= 1$, otherwise set *count* $= 0$.
  - Now, we move on P_L, starting from the left child of c_0. If the current node is c' and $c'.A < c_1.A$, we move to its right child and discard its left subtree, since all the nodes in the left subtree will have values smaller than $c_1.A$ and thus need not be counted. Otherwise if $c'.A > c_1.A$, we increase the *count* by 1 if $c'.B > c.B$ and we insert its right child (if exists) in the set S, because all the nodes in its subtree lie between $c_1.A$ and $c_2.A$. When we reach c_1, we insert its right child (if exists) in the set S.
  - We move on P_R similarly, starting from right child of c_0, but this time we insert the left child of c' (if exists) in S if $c'.A < c_2.A$, increase the *count* by 1 if $c'.B > c.B$ and move to its right child. Similarly we discard the right subtree and move to left child if $c'.A > c_2.A$. When we reach c_2, we insert its left child (if exists) in the set S. Also, we increase the *count* by 1 if $c_2.B > c.B$.

  - Now for each node s in the set S, we will count the number of nodes in the subtree rooted at s, which intersect with c. This we can do simply by applying binary search in the array $s.B\_arr$. We will find the index of the smallest element larger than $c.B$ in this array and all the elements from that index to the end of the array will have higher B value than c. We will include these many nodes in the *count* variable (i.e., increase the *count* by these many number of nodes).
  - Note that all the nodes we have included in the *count* have A value between $c.A$ and $c.B$ and have a B value higher than $c.B$, so they satisfy the intersection condition. Therefore count should contain the number of intersections for c.
  - Also, note that we do not need to explicitly maintain a set S, we can count the intersections directly instead when we are inserting the node in the set.

- We can count the number of intersections for each c in the above mentioned way, and sum of all the counts would give us the required answer.

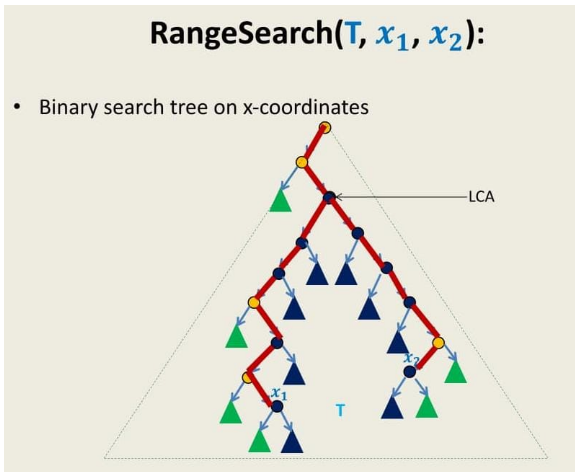
RangeSearch(T, x_1, x_2):
• Binary search tree on x-coordinates
LCA
x_1
x_2
T

Figure 1: Tree depicting the path we need to traverse. Blue coloured nodes (and subtrees) are the ones we check for intersections. Idea used is similar to the *Orthogonal Range Search problem* discussed in lectures. *Image credits :* Lecture slides

Pseudocodes for the algorithm :

We will assume that we have created the *Chords* array using the input points and sorted it in increasing order of A values. For simplicity, we will assume that all points given to us are distinct. We will assume that we have the following functions available -

- **CreateBSTfromArray($A[\,], n$) :** Takes a sorted array $A[\,]$ of size n as input, and returns a balanced BST in $O(n)$ time.
- **Merge($A[\,], B[\,], n, m$) :** Takes two sorted arrays $A[\,]$ and $B[\,]$ of sizes n and m respectively, and merges them to form a single sorted array C of size $n+m$. Returns the array C in $O(n+m)$ time.
- **Predecessor(T, x) :** Takes a balanced BST T and a number x as input and returns the predecessor of x in T, i.e., largest number smaller than or equal to x in T in $O(log(n))$ time.

We will further assume that the above functions are modified appropriately to work for our augmented BST, i.e., all comparisons in **CreateBSTfromArray** and **Predecessor** functions are based on the A values of the nodes, i.e., a node p is considered *smaller* than node q, if $p.A < q.A$. Assume 0-based indexing everywhere.

```
FillB_Arrays($root$){
    if ($root.left \neq NULL$) FillB_Arrays($root.left$);
    if ($root.right \neq NULL$) FillB_Arrays($root.right$);
    $C[\,] = \{root.B\}$; // Create an array with a single element $root.B$
    $lsize \leftarrow 0$;
    $rsize \leftarrow 0$;
    if ($root.left \neq NULL$) {
        $C \leftarrow$ Merge($root.left.B\_arr, C, root.left.size, 1$);
        $lsize \leftarrow root.left.size$;
    }
    if ($root.right \neq NULL$) {
        $C \leftarrow$ Merge($root.right.B\_arr, C, root.right.size, 1 + lsize$);
        $rsize \leftarrow root.right.size$;
    }
    $root.size \leftarrow 1 + lsize + rsize$;
    $root.B\_arr \leftarrow C$;
}

FirstLargerValue($A[\,], n, x$){
    $l \leftarrow 0$; $r \leftarrow n - 1$;
    $last\_mid \leftarrow n - 1$;
    while ($l \leq r$) {
        $mid = (l + r)/2$;
        if ($A[mid] \leq x$) $l = mid + 1$;
        else { $r = mid - 1$; $last\_mid = mid$; }
    }
    return $last\_mid$;
}

TraverseLeftPath($c, L, B\_val$){
    if($c == NULL$) return 0;
    $count \leftarrow 0$;
    if ($c.A < L$) $count \leftarrow$ TraverseLeftPath($c.right, L, B\_val$);
    else {
        $rchild \leftarrow c.right$;
        $idx \leftarrow$ FirstLargerValue($rchild.B\_arr, rchild.size, B\_val$);
        $count \leftarrow rchild.size - idx$;
        if ($c.B > B\_val$) $count \leftarrow count + 1$;
        if ($c.A > L$) $count \leftarrow count +$ TraverseLeftPath($c.left, L, B\_val$);
    }
    return $count$;
}
```

```
**TraverseRightPath**($c, R, B\_val$){
    **if**($c == NULL$) **return** 0;
    $count \leftarrow 0$;
    **if** ($c.A > R$) $count \leftarrow$ **TraverseRightPath**($c.left, R, B\_val$);
    **else** {
        $lchild \leftarrow c.left$;
        $idx \leftarrow$ **FirstLargerValue**($lchild.B\_arr, lchild.size, B\_val$);
        $count \leftarrow lchild.size - idx$;
        **if** ($c.B > B\_val$) $count \leftarrow count + 1$;
        **if** ($c.A < R$) $count \leftarrow count +$ **TraverseRightPath**($c.right, R, B\_val$);
    }
    **return** $count$;
}

**CountIntersections**($T, c$){
    // $L = c_1.A,\ R = c_2.A$, i.e. the range in which we want to search.
    $L \leftarrow c.A$;
    $R \leftarrow$ **Predecessor**($T, c.B$);
    $c_0 \leftarrow T.root$;
    **while** ($c_0 \neq NULL$){
        **if** ($c_0.A < L$) $c_0 \leftarrow c_0.right$;
        **else if** ($c_0.A > R$) $c_0 \leftarrow c_0.left$;
        **else break**;
    }
    // At this point $c_0$ must be the $LCA$ of the two nodes with $A$ values $L$ and $R$
    $count \leftarrow 0$;
    **if** ($c_0.B > c.B$) $count \leftarrow count + 1$;
    **if** ($c_0.A > L$) $count \leftarrow count +$ **TraverseLeftPath**($c_0.left, L, c.B$);
    **if** ($c_0.A < R$) $count \leftarrow count +$ **TraverseRightPath**($c_0.right, R, c.B$);
    **return** $count$;
}

**CountAllIntersections**($Chords[\,], n$){
    $T \leftarrow$ **CreateBSTfromArray**($Chords, n$);
    **FillB_Arrays**($T.root$);
    $count \leftarrow 0$;
    **for** ($i = 0$ **to** $n - 1$) {
        $count \leftarrow count +$ **CountIntersections**($T, Chords[i]$);
    }
    **return** $count$;
}
```

Space Complexity Analysis :

We have total number of nodes = n. Label all the nodes as $C_1, C_2, ..., C_n$. Consider subtree rooted at C_i. Let its size be n_i. Space occupied by $C_i = c_0 + c_1 n_i$ (each node contains some constant size variables and an array of size n_i) where c_0 and c_1 are constants. Total space occupied by our data structure $= \sum_{i=1}^{n}(c_0 + c_1 n_i) = nc_0 + c_1 \sum_{i=1}^{n} n_i$.
Consider the nodes at a particular level i. Let these nodes be $l_i, l_i + 1, ..., r_i$. Let the total space occupied by all the nodes on level i be $N(i)$.
Clearly, $N(i) = \sum_{k=l_i}^{r_i} n_k =$ (number of nodes whose level is greater than or equal to i) $< n \Rightarrow N(i) < n$. Summing over all levels, $\sum_{k=1}^{h} N(k) = \sum_{k=1}^{h} \sum_{i=l_k}^{r_k} n_i = \sum_{i=1}^{n} n_i$ (Because each node occurs in exactly one level). Here h is the number of levels or the height of the BST.
So, we have $\sum_{i=1}^{n} n_i = \sum_{k=1}^{h} N(k) < hn$. Since we have a balanced BST, $h = O(log(n))$. Therefore $\sum_{i=1}^{n} n_i = O(nlog(n))$. Therefore total space occupied by our data structure $= \sum_{i=1}^{n}(c_0 + c_1 n_i) = nc_0 + c_1 \sum_{i=1}^{n} n_i = O(nlog(n))$.

Time Complexity Analysis :

- **CreateBSTfromArray** : $O(n)$ time.
- **Merge** : $O(n + m)$ time
- **Predecessor** : $O(log(n))$ time.
- **FillB_Arrays** : Let the time taken be $T(n)$. Since our BST is balanced, both left and right subtrees have atmost $n/2$ nodes. We make 2 recursive calls, one for each child. Merge operations take $O(n)$ time. Therefore, $T(n) = O(n) + 2T(n/2) \Rightarrow T(n) = O(nlog(n))$.
- **FirstLargerValue** : Simple binary search, therefore $O(log(n))$.
- **TraverseLeftPath** : Let the time taken by this function be $T(n)$, where n = number of nodes in the BST. Since the BST is balanced, each child will have atmost $n/2$ nodes. In worst case, one call of this function calls the **FirstLargerValue** function, which takes $O(log(n))$ time, and one recursive call from a child node. All other operations take $O(1)$ time. Therefore $T(n) = O(log(n)) + T(n/2) \Rightarrow T(n) = O(log^2(n))$.
- **TraverseRightPath** : $O(log^2(n))$. Same analysis as **TraverseLeftPath**.
- **CountIntersections** : One call of **Predecessor** takes $O(log(n))$ time. In each iteration of the **while** loop we move atleast one level down in the BST. Since there are atmost $O(log(n))$ levels, it takes $O(log(n))$ time in the worst case. In the worst case, one call each of **TraverseLeftPath** and **TraverseRightPath** is made, each of which takes $O(log^2(n))$ time. Therefore, overall time taken by this function is $O(log^2(n))$.
- **CountAllIntersections** : One call of **CreateBSTfromArray** takes $O(n)$ time. One call of **FillB_Arrays** takes $O(nlog(n))$ time. Each iteration of the **for** loop makes one call of **CountIntersection**, so it takes $O(log^2(n))$ time. So the **for** loop takes $O(nlog^2(n))$ time. Overall this function will take $O(nlog^2(n))$ time.
- Converting a point to its polar form can be done in $O(1)$ time, so arrays $A[\,]$ and $B[\,]$ can be created in $O(n)$ time, modifying the arrays so that $A[i] < B[i]$ can also be done in a single scan of the array, so this also takes $O(n)$ time.

- Then $Chords[\,]$ array can be initialised using arrays $A[\,]$ and $B[\,]$ in a single scan of the arrays, so it can be done in $O(n)$ time.
- Sorting the $Chords[\,]$ array takes $O(nlog(n))$ time.
- Finally to solve the problem, we need to make one function call of **CountAllIntersections** which would take $O(nlog^2(n))$ time.

Therefore overall worst case time complexity of the algorithm is $O(nlog^2(n))$.

Dhruv Gupta: 220361

Question 2

(a) Online Algorithm for 2D Non-Dominated Points

Definition of Dominance :

A point $A = (a_x, a_y)$ is said to dominated by point $B = (b_x, b_y)$ iff $a_x < b_x$ and $a_y < b_y$.

Idea of Algorithm :

We will maintain a balanced augmented BST T (for eg., a Red Black Tree) to keep track of the non dominated points. For the i^{th} point, we will check if it is dominated by any other point in T.
If yes, then this point can not be a non-dominated point, therefore, it is not added in T.
If no, then this point is a non-dominating point among the set of i points encountered so far, so it is added in T. Also, we will remove all the points from T which are dominated by the i^{th} point.
Since, we are checking only for points in T to determine if i^{th} point is non-dominated, this is not trivial, so we will provide a justification for this.

Claim : *At each point of time, all the points in T are non-dominated points. Also, for every point p (encountered so far) not in T, there exists a point in T which dominates p.*

Proof : We will prove this by induction on number of points i. Let the points be $p_1, ..., p_i$. Base case, $i = 1$, is trivially true, since the first point is always non-dominating hence inserted in the tree. By induction hypothesis, we know that T contains all the non-dominated points among $p_1, ..., p_{i-1}$. So we only need to check if p_i is checked properly. We need to prove that p_i is inserted in the tree if and only if it is non dominated. If we find any point in T, which dominates p_i, then p_i cannot be a non dominated point and it is discarded. If we do not find any such point in T, then we need to argue that all the points p_k $(k < i)$ which are not in T also cannot dominate p. Let us assume contrary, that we have such a point p_k $(k < i)$ which is not in T, but $x_{p_k} > x_{p_i}$ and $y_{p_k} > y_{p_i}$. Now we know that no point in T dominates p_i, so for all p in T, either $x_p < x_{p_i}$ or $y_p < y_{p_i}$. This implies that for all p in T either $x_p < x_{p_k}$ or $y_p < y_{p_k}$, i.e., no p in T dominates p_k. But by the induction hypothesis, for every point p_k not in T, there must be a point in T, which dominates p_k. This is a contradiction. Therefore, if we do not find any point in T which dominates p_i, then p_i is a non dominated point among $p_1, ...p_i$, and thus must be included in T and is correctly inserted in T by our algorithm. Also, all the points which are now dominated by p_i are removed. Hence after checking i points, all the points in T are non dominated and all the points not in T are dominated by some point in T. This completes our proof.

Comparing two points : **We define a point $A = (a_x, a_y)$ to be smaller than point $B = (b_x, b_y)$, i.e., $A < B$ if $(a_x < b_x)$ or $(a_x = b_x$ and $a_y > b_y)$.**

Lemma : *If we have a list of n* ***non-dominated*** *points $\{p_1, ..., p_n\}$ sorted in increasing order, then the x coordinates are sorted in a non decreasing order, and the y coordinates are sorted in a non increasing order.*

Proof : Take any two i, j $(i < j)$. Since points are sorted in increasing order $p_i < p_j \Rightarrow x_{p_i} < x_{p_j}$ or $x_{p_i} = x_{p_j} \Rightarrow$ their x coordinates are in non decreasing order. Now, if $x_{p_i} < x_{p_j}$ and we know that p_i is non dominated, then $y_{p_i} \geq y_{p_j}$. Otherwise if $x_{p_i} = x_{p_j}$, then by definition of comparison of points, $y_{p_i} > y_{p_j}$. Therefore, for all $i, j (i < j)$, we have $x_{p_i} \leq x_{p_j}$ and $y_{p_i} \geq y_{p_j}$ Therefore, we can say that all x coordinates are sorted in a non decreasing order and all y coordinates are sorted in non increasing order.

Detailed Description of the Algorithm :

We have already seen the idea of the algorithm and that it works correctly. Now we need to provide an implementation which is efficient. Each node in the BST T represents a point and stores the following data - x, y coordinates of the point, $left, right$ pointers pointing to the left and right child of the node respectively. It may also contain some additional fields like *colour* if we are using a Red Black Tree (any self balancing tree would work).
Points in BST T are maintained in increasing order i.e., inorder traversal of the tree traverses nodes in the increasing order (refer to the definition of comparison of two points).
To ensure that for each point, we always have a successor and a predecessor in T, we insert two points $(\infty, -\infty)$ and $(-\infty, \infty)$ in T. Note that these two points are always non dominated.
When we receive the point p_i, we do the following -

- First we check, if p_i is already present in T. If yes, we simply discard it. This way we can handle duplicate points.
- Then we check if p_i is dominated by any point in T. Clearly every point p in T which is smaller than p_i, has $x_p \leq x_{p_i}$. So, no point in T smaller than p_i can dominate p_i. We only have to check for points in T larger than p_i.
- We find the inorder successor s_i of p_i, i.e., smallest point in T larger than p_i. So $p_i < s_i$.
  - If $x_{s_i} == x_{p_i}$, then $y_{s_i} < y_{p_i}$ (because $p_i < s_i$). Note that s_i does not dominate p_i. Also by *Lemma*, we can say that every point p in T which is greater than s_i has $y_p \leq y_{s_i} < y_{p_i}$. So no point larger than s_i can dominate p_i, so we insert p_i in T.
  - If $x_{s_i} > x_{p_i}$, then we compare the y coordinates. If $y_{s_i} > y_{p_i}$, then s_i dominates p_i, so we discard p_i. Otherwise, if $y_{s_i} \leq y_{p_i}$, by the similar arguement in the previous case ($x_{s_i} == x_{p_i}$ case), we can say that no point in T larger than p_i can dominate p_i, so we insert p_i in T.
- Now, we only need to check if p_i dominates any points in T. Such points need to be deleted. We know that every point p in T that has $x_p \geq x_{p_i}$, cannot be dominated by p_i. So we need to check only for points p such that $x_p < x_{p_i}$.
  We define the $x - predecessor$ of a point p_i as the largest point p with $x_p < x_{p_i}$.
  We will find the $x - predecessor$ $pred_i$ of p_i. If $y_{pred_i} < y_{p_i}$ then p_i dominates $pred_i$, so we remove $pred_i$ from T. We repeat this process until we get a $pred_i$ such that $y_{pred_i} \geq y_{p_i}$. Since we have $(-\infty, \infty)$ in T, we will always get such a $pred_i$. Clearly, $pred_i$ is not dominated by p_i. Also, by *Lemma*, every point p in T smaller than $pred_i$ will have $y_p \geq y_{pred_i} \geq y_{p_i}$, so p_i cannot dominate any of these points. At this point we can say that we have removed all the points from T which are dominated by p_i.

Pseudocodes for the Algorithm :

We assume that we have a self balancing BST data structure (such as a Red Black Tree) with following functions available -

- $T.insert(q)$: Inserts the node q in the tree T.
- $T.delete(q)$: Deletes the node q in the tree T.
- $T.predecessorX(q)$: Returns the largest node in T which has a smaller x coordinate than q, i.e., the $x - predecessor$ of q in T. This can be done easily since all the nodes in T are in non decreasing order of x coordinates (see *Lemma*).
- $T.successor(q)$: Returns the node which is inorder successor of the node q in the tree T.
- $T.search(q)$: Searches for the node in T, returns 1 if found, 0 otherwise.
- **InorderTrav(T)** : Returns the inorder traversal of nodes in BST T as a list.

All the above functions take $O(log(n))$ time, except **InorderTrav** which takes $O(n)$ time.
We will also assume that the points are available to us as an array $A[\,]$ of size n,and $A[i].x, A[i].y$ give the x and y coordinates of the point $A[i]$ respectively.

```
NonDominated2D($A[\,], n$){
    $T \leftarrow$ CreateEmptyTree();
    $T.insert((-\infty, \infty))$;
    $T.insert((\infty, -\infty))$;
    for ($i = 1$ to $n$) {
        $q \leftarrow A[i]$;
        if ($T.search(q) == 1$) continue;
        $succ \leftarrow T.successor(q)$;
        if ($succ.x > q.x$ and $succ.y > q.y$) continue;
        else $T.insert(q)$;
        $pred \leftarrow T.predecessorX(q)$;
        while($pred.y < q.y$) {
            $T.delete(pred)$;
            $pred \leftarrow T.predecessorX(q)$;
        }
    }
    $L \leftarrow$ InorderTrav($T$);
    return $L$;
}
```

Time Complexity Analysis :

We need one call of **NonDominated2D** function.
The two insertions will take $O(log(n))$ time. One call of **InorderTrav** will take $O(n)$ time. All the operations inside the **for** loop, i.e., search, successor, predecessorX, delete, insert, take $O(log(k_i))$ time, where k_i is the number of nodes in T in i^{th} iteration of the **for** loop. Also, let us assume that the **while** loop runs for n_i times in the i^{th} iteration of the **for** loop. Therefore, time taken in the i^{th} iteration of the **for** loop, $t_i = c_0log(k_i) + c_1n_ilog(k_i) < c_0log(i) + c_1n_ilog(i)$ (since, we have checked only i points so far, number of points in T, $k_i \leq i$). Here, c_0, c_1 are some constants. Now, total time for i iterations of the **for** loop,
$T(i) = \sum_{r=0}^{i} t_r = \sum_{r=0}^{i}(c_0log(r)+c_1n_rlog(r)) < \sum_{r=0}^{i}(c_0log(i)+c_1n_rlog(i)) = (c_0i+c_1\sum_{r=0}^{i} n_r)log(i)$.
Now note that $\sum_{r=0}^{i} n_r$ is the total number of iterations of the **while** loop till i points are processed,

and in each iteration of while loop, exactly one point is deleted from T. Since each point is inserted atmost once in T, total number of nodes in T can be atmost $(i + 2)$ (including the two initial insertions; note that these two points are never removed from T). So, total number of deletions (and insertions) after processing i points, can be atmost i. Therefore, $\sum_{r=0}^{i} n_r \leq i$. Therefore, $T(i) = (c_0 i + c_1 \sum_{r=0}^{i} n_r)log(i) \leq (c_0 i + c_1 i)log(i) = (c_0 + c_1)ilog(i) = O(ilog(i))$. Hence, it is ensured that processing i points takes $O(ilog(i))$ time.

(b) Algorithm for 3D Non-Dominated Points

Definition of Dominance :

A point $A = (a_x, a_y, a_z)$ is said to dominated by point $B = (b_x, b_y, b_z)$ iff $a_x < b_x$ and $a_y < b_y$ and $a_z < b_z$.

Comparing two points in 3D: **We will call a point $A = (a_x, a_y, a_z)$ to be larger than point $B = (b_x, b_y, b_z)$, i.e., $A > B$ if ($a_z > b_z$) or ($a_z == b_z$ and $a_y < b_y$) or ($a_z == b_z$ and $a_y == b_y$ and $a_x < b_x$).**

Idea of Algorithm :

We are given n points. We will sort these in non increasing order (based on above comparison definition). Let us call these $p_1, p_2, ..., p_n$. We will process these points one by one from index $i = 1$ to n. For each point p_i, we know that $z_{p_i} \leq z_{p_k}$, for all $k < i$ and $z_{p_i} \geq z_{p_k}$, for all $k > i$. Therefore, no $p_k, k > i$ can dominate p_i. So we only need to check if any point $p_k, k < i$ dominates p_i. For a point $p_k, k < i$, if $z_{p_k} > z_{p_i}$, then we only need to check if the projection of p_i on xy plane is dominated by the projection of p_k on xy plane. If yes, then p_i is dominated by p_k, otherwise it is not. On the other hand, if $z_{p_k} == z_{p_i}$, p_i is a not dominated by p_k. In this case if we look at the xy projections of p_i and p_k, because of the way we have ordered the points, we know that either $y_{p_k} < y_{p_i}$ or $x_{p_k} < x_{p_i}$. So the xy projection of p_i is not dominated by xy projection of p_k. Therefore, we can say that for any $k < i$, the projection of p_i in xy plane is dominated by projection of p_k in xy plane iff p_i is dominated by p_k. So for checking if p_i is dominated by any $p_k, k < i$, we can maintain the same data structure T we used for *Non-Dominated2D problem* and give the projections of points on the xy plane as input. However, note that, unlike the 2D problem, the set of points which are non dominated, is not the same as the points which we will keep in T. This is because, any point whose projection on xy plane is dominated by that of some other point is removed from T, but it is not removed from the set of non dominated points (if it was present there). So we will maintain a separate list to keep track of non dominated points.
Note that the comparison used for maintaining points in T was different from the comparison used to sort the given array initially.

Pseudocode for the Algorithm :

We will assume all the functions from the previous problem for 2D points. We will also assume that the points are available to us as an array $A[\ ]$ of size n, and $A[i].x$, $A[i].y$ and $A[i].z$ give the x, y and z coordinates of the point $A[i]$ respectively. Since we are given a *set* of n points we can assume that there are no duplicate points.

```
**NonDominated3D**($A[\,], n$){
    **sort**($A, n$) // sort according to the comparison we defined for 3D points
    $S \leftarrow$ **CreateEmptySet()**; // set to maintain list of non dominated points
    $T \leftarrow$ **CreateEmptyTree()**;
    $T.insert((-\infty, \infty))$;
    $T.insert((\infty, -\infty))$;
    **for** ($i = 1$ **to** $n$) {
        $q \leftarrow \{A[i].x, A[i].y\}$; // projection of $A[i]$ on the $xy$ plane
        $succ \leftarrow T.successor(q)$;
        **if** ($succ.x > q.x$ **and** $succ.y > q.y$) **continue**;
        **else** { $T.insert(q)$; $S.insert(A[i])$; }
        $pred \leftarrow T.predecessorX(q)$;
        **while**($pred.y < q.y$) {
            $T.delete(pred)$;
            $pred \leftarrow T.predecessorX(q)$;
        }
    }
    **return** $S$;
}
```

Time Complexity Analysis :

Sorting takes $O(nlog(n))$ time. Inserting an element in the set can be done in $O(log(n))$ time (if efficiently implemented using RBT; instead of a set, if we just maintain a list, it will take $O(1)$ time). There will be atmost n insertions in the set, therefore, it will take $O(nlog(n))$ time. Rest of the code is same as **NonDominated2D** function, and thus, for n points it will take $O(nlog(n))$ time (refer to time complexity analysis for 2(a)). So overall time complexity will be $O(nlog(n))$.