CS345: Assignment 1

Pragati Agrawal (220779)

August 2024

Pragati Agrawal: 220779 (Question 1)

In this question, we are given a set of n chords on a unit circle, represented as two sets $\{p_1, p_2, ..., p_n\}$ & $\{q_1, q_2, ..., q_n\}$ of n points each and we wish to find the number of intersections among these chords. Without loss of generality, let the circle be centred at the origin O.
For any point x_i on the circle, we define its **angle** θ_{x_i} as the angle formed between the line joining O and x_i and the positive x-axis, measured anti-clockwise. We can represent each chord as a pair of angles: (α, β), where α is the smaller angle out of both angles of the chord (θ_{p_i} and θ_{q_i}), and β the other one. More formally,

$$\alpha_i = min(\theta_{p_i}, \theta_{q_i}),\ \beta_i = max(\theta_{p_i}, \theta_{q_i})$$

$$chord_i = \{\alpha_i, \beta_i\}$$

We know that any point on a circle can be uniquely represented in *Eulerian form* as $r.e^{i\theta}$. Here since all points lie on the same circle, all have the same $r = 1$. Therefore, each point x_i can be uniquely represented by its angle θ_{x_i}.

Condition of intersection in terms of angle θ_{x_i}:
Consider two chords A and B having end-points: $\{p_i, q_i\}$ and $\{p_j, q_j\}$, respectively. Without loss of generality, let $\theta_{p_i} < \theta_{q_i}$ and $\theta_{p_j} < \theta_{q_j}$. Therefore, $A = \{\theta_{p_i}, \theta_{q_i}\}$ and $B = \{\theta_{p_j}, \theta_{q_j}\}$. Also, without loss of generality, assume $\theta_{p_i} < \theta_{p_j}$. Now, **A and B will intersect, iff** $\theta_{p_j} < \theta_{q_i}$ **and** $\theta_{q_j} > \theta_{q_i}$.

Description of the Data Structure used:

We keep an **Augmented Balanced Binary Search Tree** for the problem. The BST would be made according to the α_i of each chord. Each node of the tree would contain:

- **val:** The α of that node.
- **b_val:** The β of that node.
- **left:** The pointer to the left child of the node.
- **right:** The pointer to the right child of the node.
- **size:** The size of the subtree rooted at that node.
- **arr:** Array containing the sorted order of β_i of each chord in the subtree rooted at that node.

We use a BST, which essentially is a divide-and-conquer algorithm paradigm.

Description of the Algorithm:

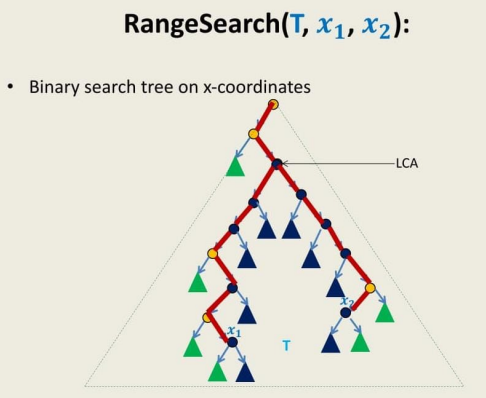
Algorithm for building the augmented BST data structure:

- We will first create an **array containing each chord as its pair of angles**, let's call it *chords*. We will then sort it in ascending order of α_i.
- Now we will create a BST called **Tree** of the array *chords* recursively, using top-down approach.
  - We will assign the **val** and **b_val** of each node directly while adding the new node.
  - The **left** and **right** pointers would point to the return value nodes of the recursive calls of the left and right portions respectively. For leaf nodes, they would be *NULL*.
  - The **size** would be assigned as $size(node) = size(left\_child) + size(right\_child) + 1$. For NULL nodes, it would be *0*, while for leaf nodes, it would be *1*.
  - The **arr** would be the merge of the two *arr* of the left and right children, along with its own *b_val* inserted. We know by induction that the *arr* of the left and right children would be sorted, so to create the array for the current node, we will merge the two sorted arrays of its children, and also insert its *b_val* appropriately. For leaf nodes, the array would contain just one element, its own *b_val*.

Algorithm for finding the number of intersections for each chord:

To find the number of intersections, for any chord in *chords[i]*, we would find the chords having α such that $\alpha_i < \alpha < \beta_i$. This can be done as:

- Find the inorder successor $succ_i$ of α_i. (The smallest node with $val \geq \alpha_i$.)
- Find the inorder predecessor $pred_i$ of β_i. (The largest node with $val \leq \beta_i$.)
- Now we perform $RangeSearch(Tree, succ_i, pred_i)$ to find all the nodes who themselves and/or their some subtree lies within this range (navy-blue colored in the figure below).



Now, for all the subtrees found by *RangeSearch*, we would find the position of β_i in its **arr** field. Subtracting it from the **size** field of that node would give us the number of nodes having β value greater than β_i. We had already ensured that each node in the subtree has α value between α_i and β_i. So all these chords intersect with *chords[i]* and hence this number will be added to the answer. For the nodes themselves who lie within the range, but not their subtree (i.e. the **blue nodes** in the figure), we would check simply if its β value is greater than β_i or not. Accordingly, we would increment the answer. (similar to the idea discussed in **Lecture 5**)

Pseudo-codes for the Algorithm:

Pseudocode for building the augmented BST data structure:

(**Node*** is the struct for our nodes of the tree.)

```
Node* createTree (chords[ ], left_index, right_index){  // build tree
    if(left_index > right_index) return NULL;
    Node* ret_node = new Node();
    if(left_index == right_index){  // leaf nodes
        ret_node->val = chords[left_index].alpha;
        ret_node->b_val = chords[left_index].beta;
        ret_node->left = ret_node->right = NULL;
        ret_node->size = 1;
        ret_node->arr = {chords[left_index].beta};
        return ret_node;
    }
    mid_index = (left_index + right_index) / 2;
    ret_node->val = chords[mid_index].alpha;
    ret_node->b_val = chords[mid_index].beta;
    ret_node->left = createTree(chords, left_index, mid_index-1);
    ret_node->right = createTree(chords, mid_index+1, right_index);
    ret_node->size = ret_node->left->size + ret_node->right->size + 1;
    ret_node->arr = merge (chords[mid_index].beta, ret_node->left->arr,
                    ret_node->right->arr, ret_node->left->size, ret_node->right->size);
    return ret_node;
}

merge (val, arr1[ ], arr2[ ], n, m){   // takes two sorted arrays and merges them
    // Create a new array called ret_arr of size (n+m+1)
    i=0, j=0, k=0, flag=0;
    while(i < n && j < m){
        if(flag==0 && val < arr1[i] && val < arr2[j]) {
            flag = 1;
            ret_arr[k++] = val;
        }
        else if(arr1[i] <= arr2[j]){
            ret_arr[k++] = arr1[i++];
        }
        else ret_arr[k++] = arr2[j++];
    }
    while(i < n){
        if(flag==0 && val < arr1[i]) {
            flag = 1;
            ret_arr[k++] = val;
        }
        else ret_arr[k++] = arr1[i++];
    }
```

```
    while(j < m){
        if(flag==0 && val < arr2[j]) {
            flag = 1;
            ret_arr[k++] = val;
        }
        else ret_arr[k++] = arr2[j++];
    }
    if(flag == 0) ret_arr[k++] = val;
}
```

Pseudocode for counting the number of intersections:

```
Node* Tree = createTree (chords, 0, n); (assuming n is the total no of chords)

int find_intersections (chords[ ], n){  // finds total no of intersections
    total_count = 0;
    for (i = 0; i < n; i++){
        succ_i = find_succ (Tree, chords[i].alpha);
        pred_i = find_pred (Tree, chords[i].beta);
        total_count += countRangeSearch (Tree, succ_i, pred_i, chords[i].beta);
    }
    return total_count;
}

int countRangeSearch(Tree_root, succ, pred, beta){ //counts intersections in
    count = 0, foundLCA=0;                          //the interval (succ to pred)
    Node* temp = Tree_root;
    Node* left_ptr = NULL; Node* right_ptr = NULL;
    if(succ->val < temp->val && temp->val < pred->val) {
        foundLCA = 1;
        left_ptr = temp->left;
        right_ptr = temp->right;
    }
    while( ! foundLCA){
        if(succ->val < temp->val && temp->val < pred->val) {
            foundLCA = 1;
            left_ptr = temp->left;  right_ptr = temp->right;
        }
        else if(succ->val < temp->val && pred->val < temp->val)
            temp = temp->left;
        else if(succ->val > temp->val && pred->val > temp->val)
            temp = temp->right;
    }
    if(temp->b_val > beta) count += 1;
    count += countLeftSearch(Tree_root, left_ptr, succ, beta);
    count += countRightSearch(Tree_root, right_ptr, pred, beta);
    return count;
}
```

```
int countLeftSearch (Tree_root, ptr, succ, beta){  // counts intersections in the
    count = 0, found=0;                           //path from LCA to succ(alpha_i)
    while(ptr != succ){
        if(succ->val > ptr->val)
            ptr = ptr->right;
        else{
            if(ptr->b_val > beta) count += 1;
            rank = findRank (ptr->right->arr, ptr->right->size, beta);
            count += ptr->right->size - rank;
            ptr = ptr->left;
        }
    }
    if(succ->b_val > beta) count += 1;
    rank = findRank (succ->right->arr, succ->right->size, beta);
    count += succ->right->size - rank;
    return count;
}

int countRightSearch (Tree_root, ptr, pred, beta){ // counts intersections in the
    count = 0, found=0;                             //path from LCA to pred(beta_i)
    while(ptr != pred){
        if(pred->val < ptr->val)
            ptr = ptr->left;
        else{
            if(ptr->b_val > beta) count += 1;
            rank = findRank (ptr->left->arr, ptr->left->size, beta);
            count += ptr->left->size - rank;
            ptr = ptr->right;
        }
    }
    if(pred->b_val > beta) count += 1;
    rank = findRank (pred->left->arr, pred->left->size, beta);
    count += pred->left->size - rank;
    return count;
}

int findRank (b_array[ ], size, beta){  // finds largest index in the array
    left = 0, right = size;             // having value <= beta
    while(left <= right){
        mid = (left + right) / 2;
        if(b_array[mid] <= beta) {
            ret_val = mid; left = mid + 1;
        }
        else right = mid - 1;
    }
    return ret_val;
}
```

Time Complexity Analysis:

We know that converting any point from (p_i, q_i) form to $(\theta_{p_i}, \theta_{q_i})$ form would take constant time.

For building the Tree Data Structure:

- **merge(val, arr1[ ], arr2[ ], n, m):** The merge function takes in two sorted arrays of size n and m respectively, and merges them, as well as the value *val*. This is a **linear** operation and would take $O(max(n, m))$ time.
- **createTree (chords[ ], left_index, right_index):** The create tree recursively builds the tree. It visits every node exactly once and adds it to the tree appropriately.
  First it calls the left and right subtrees, and then it merges their β arrays. We know that the merge function takes *O(size of subtree)* time. So the recurrence formed is:
  $$T(n) = 2T(n/2) + O(n) + c \Rightarrow T(n) = O(nlogn)$$

For finding the number of intersections:

- **findRank(b_array[ ], size, beta):** It is very similar to the binary search function, where it takes the sorted array *b_arr[ ]* and returns the largest index having value less than or equal to input parameter β. It works in $O(log(size))$ time.
- **countRightSearch (Tree_root, ptr, pred, beta):** It performs a binary search in the tree searching for its parameter *pred*. Along the path, on atmost every level, it calls the **findRank** function, along with some constant time operations. We know that a balanced BST will have atmost $O(logn)$ levels, and findRank also takes $O(logn)$ time.
  Therefore, it takes $O(log^2 n)$ time in the worst case.
- **countLeftSearch (Tree_root, ptr, succ, beta):** It works very similar to **countRightSearch** function with slight modifications, so it also takes $O(log^2 n)$ time.
- **countRangeSearch (Tree_root, succ, pred, beta):** It first finds the least common ancestor of the two parameters *succ* and *pred* in the tree. This takes $O(logn)$ time (since atmost $O(logn)$ levels). Then it calls the functions **countLeftSearch** and **countRightSearch**, which take $O(log^2 n)$ time each.
  Therefore this function overall takes $O(log^2 n) + O(logn) = O(log^2 n)$ time.
- **find_intersections(chords[ ], n):** It runs a loop over all chords in the input array, and finds the number of chords which intersect with this chord, and greater than it. Each iteration of the loop calls **countRangeSearch** whose time complexity is $O(log^2 n)$, so the overall time complexity of **find_intersections** is $O(n * log^2 n)$.

Hence the overall time complexity is $O(n * log^2 n)$.

Space Complexity Analysis:

Since we have made a balanced BST, it would have *O(log n)* levels for n nodes. Now, at each node, we keep 5 variables and an array storing the β values of all nodes in its subtree. We know that the β value of any node will appear in the arrays of all its ancestors upto the root. Hence each of the n nodes can come in atmost *log n* levels in the worst case. We have not used any other data structure in any function. Apart from that, all the functions use constant space.
So the overall space complexity is $O(nlogn)$.

Pragati Agrawal: 220779 (Question 2(a))

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

In this question, we will be given a set of n points in an online manner (i.e. one by one) and we need to update the set of non-dominated points accordingly. We assume the input points are in the format (x_i, y_i).

⋆ **Point P(x,y) is said to dominate over point A(a,b) if $x > a$ and $y > b$.**

⋆ **Point P(x,y) is said to be greater than point A(a,b) if $x > a$ or ($y < b$ if $x = a$).**

Description of the Data Structure used:

We maintain an **Augmented Red Black Tree** sorted according to the x coordinate values of the points. **If two points have the same x coordinates, we would store them in non-increasing order of their y coordinates.** At any point in time, it would have all the distinct non-dominating points seen so far. Each node of the tree would contain fields:

- **x_val:** The x coordinate of that point.
- **y_val:** The y coordinate of that point.
- **left:** The pointer to the left child of the node.
- **right:** The pointer to the right child of the node.
- **color:** The color of the node (red/black).

Basic Description of the Algorithm:

In the algorithm, we maintain in the tree the set of points that are all non-dominating. So, after seeing $i-1$ points, we include the i^{th} point P_i, if it is not dominated by any other point currently in the tree. Otherwise we ignore P_i. Also, if included, we need to update the tree by removing all points that are now being dominated by P_i.

Proof of correctness of the Algorithm: We will prove this by induction on number of points.
Base Case: For 1 point, we would always include it in the tree. So base case is trivially true.
Let the input points till now be $P_1, P_2, ..., P_{i-1}$. And let us assume that currently $j \leq (i-1)$ points are present in the tree. Now for any point P_i, if it is not included in the tree, it means there is a point in the tree that dominates P_i. In this case nothing happens, and by induction hypothesis, the algorithm holds for i too.
If P_i is included in the tree, then we need to prove that it was indeed not being dominated by any point from $P_1, P_2, ..., P_{i-1}$. **The algorithm includes any point P_i iff it is not dominated by any of the j points currently in the tree.** We show that checking just these j points suffices.
Assume on the contrary, there was a point P_k not in the tree that dominates P_i. This means that $x_{p_k} > x_{p_i}$ and $y_{p_k} > y_{p_i}$. Now, we know that for all j points in the tree, $x_{p_j} < x_{p_i}$ or $y_{p_j} < y_{p_i}$. Since P_k was not in the tree, by induction hypothesis for $i-1$ points, there must be some point P_m in tree that dominates P_k, i.e. $x_{p_m} > x_{p_k}$ and $y_{p_m} > y_{p_k}$. This implies $x_{p_m} > x_{p_i}$ and $y_{p_m} > y_{p_i}$, i.e. we have a point P_m in the tree that dominates P_i. This is contradiction to our assumption of

including P_i in the tree. Hence our assumption was incorrect.
After inclusion, we remove all points existing in the tree, which are being dominated by P_i, upholding correctness for i^{th} point. Hence proof is complete.

Detailed Description of the Algorithm:

Initially, the tree would be empty. The first point will always be included in the tree. Now to check for the i^{th} point P_i, we will find its **inorder successor** S_i in the tree ($S_i > P_i$). If x_i has no inorder successor, it means it has the highest x coordinate. So it cannot be dominated by any of the points seen so far. So it will be inserted in the tree. Otherwise, it does have a successor $S_i = (x_{s_i}, y_{s_i})$. Now, there can be two cases:

- $x_{s_i} > x_i$: This can further have 3 cases:
  - $y_{s_i} > y_i$: Then clearly the point P_i is dominated by S_i. Hence it will be ignored.
  - $y_{s_i} \leq y_i$: We claim that in this case P_i would not be dominated by any point seen so far, and we include it in our set.
    Proof: By definition of dominance, P_i cannot be dominated by S_i. Now, on the contrary, if any other point A_i exists in the tree that dominates P_i, it must have $x_{a_i} > x_i$ and $y_{a_i} > y_i$. By definition of successor, we can say that if $x_{a_i} > x_i$ then $x_{s_i} \leq x_{a_i}$ and we know that $y_{s_i} \leq y_i < y_{a_i}$. There are 2 cases:
    * $x_i < x_{s_i} = x_{a_i}$ and $y_{s_i} \leq y_i < y_{a_i}$: This case is not possible. This is because S_i is the inorder successor of P_i. If such an A_i exists in the tree, such that $x_i < x_{s_i} = x_{a_i}$ and $y_{s_i} < y_{a_i}$ then $A_i < S_i$, and hence A_i would be the inorder successor of P_i, which is contadictory to our assumption.
    * $x_i < x_{s_i} < x_{a_i}$ and $y_{s_i} \leq y_i < y_{a_i}$: In this case, A_i dominates S_i, which means S_i could not have been present in the tree. This is again a contradiction.

    So we find that in both cases, we arrive at a contradiction.Therefore, P_i would not be dominated and we can safely include it.

- $x_{s_i} == x_i$: In this case we must have $y_{s_i} < y_i$, by the property of inorder successor. (If $y_{s_i} == y_i$, it is a duplicate point, and we simply ignore it). We know that P_i can not be dominated by S_i or any point having the same or smaller x coordinate by definition of dominance. Now by the proof below, we establish that it will not be dominated by any point having greater x coordinate than S_i in the tree. Hence we would insert it.
  Proof by contradiction: Let there be another point $A_i = (x_{a_i}, y_{a_i})$ that dominates P_i. Therefore, we must have $x_{a_i} > x_i$ and $y_{a_i} > y_i$. Also we have $x_{a_i} > x_{s_i}$. Now there can be 2 cases:
  - $y_{a_i} \leq y_{s_i}$: By the property of inorder successor, we have $y_{s_i} < y_i$ and $y_{a_i} \leq y_{s_i}$, therefore $y_{a_i} < y_i$. Clearly, it cannot dominate P_i, so this case is not possible for our assumption.
  - $y_{a_i} > y_{s_i}$: Clearly, A_i dominates S_i, and by correctness of our algorithm, S_i could not have been present in the tree. Therefore we arrive at a contradiction.

After including P_i, we will keep deleting predecessors of x_i (nodes which have x coordinate strictly less than x_i) which are dominated by P_i, until either we get a point having a higher y coordinate or all predecessors are removed.
Claim: If predecessors of x_i remain in the tree after removals of the i^{th} step, we claim that they cannot be dominated by P_i.

Proof for the above claim: Let P_f be the last predecessor to not be removed. Consider any point $P_l < P_f$ smaller than P_i, which does not get removed. Now, if it gets dominated by P_i, then $x_i > x_{p_l}$ and $y_i > y_{p_l}$. Since P_f was not removed, $y_{p_f} > y_i$, and we know that $x_{p_l} \leq x_{p_f} < x_i$. There are two cases:

- $x_{p_l} < x_{p_f}$: Then P_l is dominated by P_f, and by correctness of the algorithm, cannot be present in the tree. So this is not possible.
- $x_{p_l} = x_{p_f}$: Since $P_l < P_f$, therefore, $y_{p_l} > y_{p_f} > y_i$. But we had assumed that P_l gets dominated by P_i. This again is a contradiction.

At the end, we would return all the nodes present in the tree by making a traversal of the tree, and storing them in a list.

Pseudo-codes for the Algorithm:

We first create an empty Red Black Tree (**T**), having struct Node* as defined above. We assume we have support for the following functions working in the mentioned time complexities (consider n to be the number of nodes in the tree at any point in time):

- **Insert(T, p):** Inserts node p into the tree, in $O(logn)$ time.
- **Delete(T, p):** Deletes node p from the tree, in $O(logn)$ time.
- **Pred(T, p):** Returns the largest point q in the tree having x_val strictly smaller than x_p, in $O(logn)$ time. If there is no predecessor, it returns *NULL*.
- **Succ(T, p):** Returns the smallest point q in the tree having x_val greater than or equal to x_p, in $O(logn)$ time. If there is no successor, it returns *NULL*.
- **Trav(T):** Returns all the nodes in the tree as a list, in $O(n)$ time.

(Assume that occasionally, we would need to do rotations and color balancing of the tree, to maintain its black height, so that all subsequent operations are complete in the mentioned times).

```
Node* Tree = createEmptyTree();

online_nonDominated_2D_points(n){  // n is the number of points in input
    for(int i=0; i<n; i++){
        x_i = points[i].x;
        y_i = points[i].y;
        p_i = createNode(x_i, y_i);
        Node* succ = Succ(Tree, p_i);  //returns the inorder successor of p_i in tree
        if((succ == NULL) or (succ->x_val == x_i) or
              (succ->x_val > x_i and succ->y_val <= y_i)) {
            insert(Tree, p_i);
            remove_dominated(Tree, p_i);
        }
    }
    return Trav(Tree);
}
```

```
remove_dominated(Tree, p_i){
    Node* pred = Pred(Tree, p_i);  // returns the largest node having x_val
                                   // strictly smaller than x_i
    while(pred != NULL && pred->y_val < p_i->y_val){
        delete(Tree, pred);
        pred = Pred(Tree, p_i);
    }
}
```

Time Complexity Analysis:

Upto the insertion of the i^{th} point, the *for* loop would have iterated linearly, i times, visiting each point exactly once. This means each point can be inserted in the tree **atmost** once. Now, upto some $j \leq i$ insertions, we can have atmost j deletions. (This is because any node once deleted, cannot be inserted again in the tree). So for i points, we can have **atmost** i **insertions and atmost** i **deletions** in the tree. Both these operations take $O(logi)$ time for i nodes. The *Pred* and *Succ* functions also take $O(logi)$ time for a balanced tree of size i.
Therefore, the overall time complexity would be $O(i * logi)$, upto any i^{th} point.

Space Complexity Analysis:

We are using a RBT data structure, which takes $O(n)$ space for n nodes. Apart from that, the functions would take constant space to execute.
Therefore, the overall space complexity would be $O(i)$, upto the i^{th} point.

Pragati Agrawal: 220779 (Question 2(b))

2(b): Algorithm for 3D Non-Dominated Points:

We assume the input points are in the format (x_i, y_i, z_i). To find the set of non-dominated points in 3D, we heavily use the algorithm used in part (a).

⋆ **Point P(x,y,z) is said to dominate over point A(a,b,c) if** $x > a$ **and** $y > b$ **and** $z > c$**.**

⋆ **Comparison Condition:** Point P(x,y,z) is said to be greater than point A(a,b,c) if $z > c$ or ($y < b$ if $z = c$) or ($x < a$ if $z = c$ and $y = b$).

Description of the Data Structure used:

We maintain the same data structure *Tree* as mentioned in part (a). We also maintain a set S of all Non-Dominated points to be returned as the answer.

Description of the Algorithm:

The *Tree* now contains all points seen so far, that are non-dominating w.r.t. their x, y coordinates only. Then we sort all the points in non-increasing order according to the above mentioned comparison condition.

For any point, P_i, we would have some $j \leq (i-1)$ points currently included in the tree, and some $u \leq (i-1)$ points included in the answer set S.

- If P_i is dominated in *Tree*, based on its x and y coordinates, then we would conclude that it is being dominated, and simply ignore it.
- Else, we would include it in the tree and in answer set S.
- We would remove the nodes from the tree now being dominated in x and y coordinates but not from the set S.
- Return S as the answer.

Assertion: After i points, the set S contains all the non-dominated points upto i, and the *Tree* contains set of all points that are non-dominated w.r.t. (x, y) coordinates.
Proof of correctness: We give a proof by induction on the number of points. Suppose we have processed upto $i-1$ points and we have some $j \leq (i-1)$ nodes in the tree. We will check for non-dominance of P_i only for the j points currently in the tree. This is because:

- For any point P_l lying after P_i in the sorted array, we know that its $z \leq z_i$. Therefore it cannot dominate P_i.
- For any point P_l between $P_1, P_2, ..., P_{i-1}$ and not in the tree that dominates P_i, we can always find a point P_m in the tree that dominates P_i. If P_l dominates P_i, then $x_{p_l} > x_i$ and $y_{p_l} > y_i$ and $z_{p_l} > z_i$. Since P_l was not present in the tree, there must be some point P_m in the tree that dominates P_l (w.r.t. $x-y$ coordinates), i.e. $x_{p_m} > x_{p_l}$ and $y_{p_m} > y_{p_l}$. Also, by the order in which we are processing points, both P_l and P_m must have z coordinate greater than z_i (if $z_{p_m} = z_{p_i}$ then $x_{p_m} > x_{p_l} > x_i$ and $y_{p_m} > y_{p_l} > y_i$, which means that P_i should have been processed before P_m in the array). Therefore P_m would also dominate P_i. Hence checking only with points present in the tree suffices.
- For any point P_l between $P_1, P_2, ..., P_{i-1}$ and present in the tree, we must have $z_{p_l} \geq z_i$. There are 2 cases:
  - $z_{p_l} > z_i$: So P_i is dominated in the z coordinate by P_l. Now if P_i was dominated in the Tree by some P_l, it means all 3 coordinates of P_i are dominated by P_l and in that case, we would not include it in the answer. On the other hand, if P_i is not dominated by any P_l, then we can say that P_i is a non-dominated point w.r.t. its x or y coordinate. So we would include it in S and in the tree. We would also suitably update the tree after including P_i.
  - $z_{p_l} = z_i$: In this case P_i is a non dominated point and should be included.
    Proof for the above claim: The order in which we are processing points ensures that the z coordinates of points are in non-increasing order. So if there is some point in the tree with the same z coordinate, it must have either x or y or both coordinates smaller than P_i. If there was some point that could dominate P_i, it must have dominated P_l, and hence P_l would not have been present in the tree. But we had assumed P_l exists in the tree. Therefore, P_i must be non-dominated.

Pseudo-codes for the Algorithm:

The function *online_nonDominated_2D_modified* maintains non-dominated points according to their x and y coordinates. It returns false if the point gets dominated in the tree, otherwise true.
We use the same function for *remove_dominated* as in part (a).

```
Node* Tree = createEmptyTree();

nonDominated_3D_points (points[ ], n){
    sort(points[ ], comp_condition);
    Set S = createEmptySet();
    for(int i=0; i<n; i++){
        if(online_nonDominated_2D_modified(points[i].x, points[i].y)==true)
            insert(S, points[i]);
    }
    return S;
}

online_nonDominated_2D_modified(x, y){
    p_i = createNode(x, y);
    Node* succ = Succ(Tree, p_i);
    if(succ == NULL or succ->x_val == x_i or
         (succ->x_val > x_i and succ->y_val <= y_i)){
           insert(Tree, p_i);
           remove_dominated(Tree, p_i);
           return true;
    }
    else return false;
}
```

Time Complexity Analysis:

Initially sorting the input points according to the comparison condition takes $O(nlogn)$ time.
Now, for any point P_i, we check if it gets dominated by any earlier point in (x, y) coordinates. We know that finding successor in a balanced tree of n nodes takes $O(logn)$ time. Rest are constant time operations in the *online_nonDominated_2D_modified* function.
Now by the same logic as stated in 2(a), we can say that each point can be inserted into the tree **atmost** once. Similarly each point can be removed from the tree atmost once. Therefore, overall for n points, we can have **atmost n insertions and n deletions**. Insertion into the set also takes $O(logn)$ time.
Therefore, we can say that overall, for n points, we will have worst case time complexity of $O(nlogn)$.

Space Complexity Analysis:

We are using a RBT data structure, which takes $O(n)$ space for n nodes. Apart from that, we have a set S which stores the answer. It also can have size $O(n)$. The functions would take constant space to execute. Therefore, the overall space complexity would be $O(n)$ for n points.