Theoretical Assignment 1

Gurpreet Singh - 150259 . Nikita Awasthi - 150453

April 2017

Question 1

Part 1

Pseudo Code

Algorithm 1 Algorithm to compute non-dominated points in 2D

```
procedure GetNonDominatedPoints(points)
    if length(points) ≤ 1 then
        return points
    end if

    median = getMedian(points)
    points_l, points_r = divide(points, median)      ▷ Function to divide the points based on a value (here median)

    ndp = findMaxYPoint(points_r)                    ▷ Function to find the point with max y value

    points_lt = ()
    for point in points_l do
        if point is not dominated by ndp then
            points_lt.push(point)
        end if
    end for

    points_rt = ()
    for point in points_r do
        if point is not dominated by ndp then
            points_rt.push(point)
        end if
    end for

    left = GetNonDominatedPoints(points_l)
    right = GetNonDominatedPoints(points_r)
    return left + right + ndp
end procedure
```

Time Complexity Analysis

If the number of points for any division is n, then we can find the median, divide the points, check for maximum y coordinate in one half as well remove points that are dominated by the maximum T coordinate point in $O(n)$.

Starting with all the points, we have n points, and in each further recursive call, the number of points will reduce by at least half (As we are dividing by the median and also removing some of the dominated points)

By looking at the length of the input points, we are also returning if there are no points, or just one point. Hence, for every leaf of the recursive tree, we must have at least one point. Therefore, we can say that the maximum height of the recursive tree is $log(n)$.

Claim: We will only end up in h leaf nodes of the recursive tree.
Proof: Suppose we have n points, out of which we find ndp and remove all points dominated by ndp. If the number of points in any division remaining is zero, clearly we ignore that division. However, if it is non-zero, we can surely say that there is a non-dominating point in that division, if not, it has to be removed by ndp. Therefore, we will proceed only if we see more than one point, hence ensuring that we end up only in h leaf nodes.

It is clear that the number of points in one division at a height $i(0 <= i <= log(n)$ will be $\frac{n}{2^i}$.
Claim: The worst case time will be observed when all the leaf nodes are found at $i = log(h)$ **Proof:** Assume this is not the case. Then, at least one leaf node will be observed at height $i > log(h)$. Since we are not going to a subtree of the recursive tree if there is no Non-Dominating Node there, then we are not losing any time on any other path. Hence the number of points now to be dealt with have been reduced by at least $\frac{n}{2^{log(h)}}$ and increased by $\frac{n}{2^i}$, but since $i > log(h)$, the total number of points have been reduced. Therefore, this is not the worst case scenario. Hence our claim holds.

Therefore, the time taken at the $i^{th}(0 <= i <= log(h))$ height of the tree is -

$$T_i = 2^i \frac{n}{2^i}$$
$$T_i = n$$

However, discarding each subtree also takes $O(1)$ time, and since we have h leaves, we are discarding h times. Therefore, the total time is $\sum T_i + h = nlog(h) + h$

Hence, the worst case time for this problem is $nlog(h)$

Part 2

Part A

Pseudo Code

Algorithm 2 Online Algorithm for non dominated points

```
T = new BinaryTree( )                          ▷ Initialize a new Empty Self-Balancing BST
                                               ▷ T will have two values x and y, but will be sorted using x only

procedure checkForNonDominating(point)
                                               ▷ Inserts point in T if dominating, and returns True if Non-Dominating else False

    head = T.insert(point)                     ▷ Head is the pointer to the node
    succ = T.getSuccessor(head)

    if succ ≠ NULL and succ.y > point.y then
        delete(head)
        return FALSE
    end if

    removeDominatedPoints(head, point)
    return TRUE
end procedure

procedure removeDominatedPoints(T, head, point)
    split_y = T.reverseSearch(head.y)          ▷ Search for element with y ¡ head.y and x ¡ head.x with left child as the larger child for every node

    if split_y = NULL then
        T1, T2 = split(T, split_y)
        T2, T3 = split(T1, head)

        T = merge(T1, T3, point)
    end if
end procedure
```

Proof of Correctness

Handling the insertion of the i^{th} point, P_i

Lemma 1: The BST sorted in increasing order of x will be sorted in decreasing order of y
Proof:
Let us assume that the balanced binary search tree will have only non-dominated points at any given instance. For any given $i < j$, $x_i < x_j$. Now since all the points in our BST are non-dominated points, y_i should be greater than y_j for all values of i and j. Therefore, BST will be sorted in decreasing order of y

In the *reverseSearch* function, we use this property to find the the point with Y coordinate just smaller than the inserted point among the elements with X coordinate smaller than the inserted point.

Lemma 2: An element will be inserted in the binary search tree if and only if it is a non-dominated point among the streamed points
Proof:
Assume that the BST only contains Non-Dominating points before inserting the i^{th} element. (proven in the

next Lemma)
For any element $P_i(x_i, y_i)$, it is not dominated by any point with X coordinate less than x_i. Therefore we only need to compare the element with it's successors in the BST (as all predecessors have X coordinate less than x_i).
For our BST, that is sorted in increasing order of x, y is sorted in decreasing order as proved above. If the direct successor of this point (S_1), does not dominate it, any k^{th} successor ($k > 1$) does not dominate it. So it is sufficient to check the direct successor for non dominatingm otherwise delete it. Hence the point is inserted if and only if it is Non Dominating

Lemma 3: If any point is dominated by some point, it is deleted from the BST, i.e. Only Non-Dominating points will persist in the BST
Proof (By Induction):
Induction Hypothesis: Assume BST only contains Non-Dominating points before inserting the i^{th} element
Base Case: When $i = 1$ or $i = 2$ (Valid in both cases)
Proof:
If the induction hypothesis is given, then Lemma 2 is valid. Hence we insert P_i only if it is Non-Dominating over the streamed points. To maintain the validity of this Lemma, we need to remove all those points from the BST which are dominated by P_i.
This is handled by the *removeDominatedPoints* function. In this, we are proceeding with three steps.
If the point P_i is not inserted in the BST (i.e. it is not a Non-Dominating point), then we are done. Otherwise, we can have two cases -
Case 1: Point P_i does not dominate any point in the BST.
If this is true, then the pointer $split_y$ will be NULL. Hence we do not need to handle this case separately.

Case 2: Point P_i dominates, say k points ($k >= 1$), P_{i_1} to P_{i_k}
Split the BST from the point which has it's Y coordinate just smaller than y_i and X coordinate smaller than x_i. Clearly the split point will be P_{i_1}. Make another split on P_i. Hence we will have three trees, $T1$, $T2$, $T3$.

T1 contains all points with Y coordinate greater than y_i, hence all Non-Dominating (not dominated by P_i

T2 contains all points with both X and Y coordinates less than those of P_i, hence all dominated by P_i, and P_i itself

T3 contains all points with X coordinate greater than x_i, hence all Non-Dominating (not dominated by P_i

Therefore, dropping all points in the tree T2 (except for P_i), removes all dominated points.
To merge the points into one tree, we simply need to merge the trees $T1$ and $T3$, while also including P_i. Since all the points in the tree $T1$ have X coordinates less than x_i and all the points in the tree $T3$ have X coordinates greater than x_i, we can simply merge $T1$ and $T2$ using P_i. Therefore the lemma holds true.

Hence by using Lemma 2 and Lemma 3, we can say that this tree always contains all Non-Dominating points that have been streamed until now.
Therefore for reporting all Non-Dominating Points, we can report all elements of this tree.

Time Complexity Analysis

Time Analysis of insertion of P_i

It is trivial that maximum number of elements in the tree can be $i - 1$. Hence all BST operations shall be $O(log(i))$

removeDominatedPoints: One Search + Two Splis + One Merge
Since all operations can be done in $O(log(i))$, the order of this function remains $O(log(i))$

Since in the function *checkForNonDominating*, we are only inserting and deleting, the order of this

function is again $O(log(i))$ (Other than calling the function *removeDominatedPonts*)

Therefore, the whole insertion, while maintaining the tree balanced and keeping all it's points Non-Dominated is done in $O(log(i))$.

Part B

We will be using the code from the previous online method in 2D to obtain all Non-Dominating points in 3D. In this case, the Tree (T) will be storing the points' X and Y coordinates.

Pseudo Code

Algorithm 3 Determine all Non-Dominating Points from a set of points

```
T = new BinaryTree( )                          ▷ Initialize a new Empty Self-Balancing BST

procedure getNonDominatingPoints(points)       ▷ Points is a list/array of points

    points = sort(key = −point.z)              ▷ Sort in Descending order of Z coordinate
    nonDominatingPoints = new List( )

    for point in points do
        isDominating = checkForNonDominating(point)   ▷ Used from previous part i.e. Part A

        if isDominating then
            nonDominating.push(point)
        end if
    end for

    return nonDominatingPoints
end procedure
```

Proof Of Correctness

We have a list of points, which are sorted in descending order of their Z coordinates. We are inserting the points in an online fashion (starting from the first in the sorted list) in the Binary Search Tree which contains the Non-Dominating points (from this list) when projected on the x-y plane.

Lemma 1: A point is a Non-Dominating point iff it is inserted in the BST (might get deleted later), i.e. if the call on the function *checkForNonDominating* returns TRUE, then the point is a Non-Dominating point

Proof:

Let us analyze a point with index i

Since the points are sorted in descending order on their Z coordinates, we know $points[i].z > points[j].z$ $(\forall j > i)$

Hence, point i cannot be dominated by any point that has a higher index than i. Therefore we don't care what happens to the point (in the BST) once we move on to the next index.

We know $points[i].z < points[j].z$ $(\forall j < i)$. Hence point i is Non-Dominating iff it $points[i].y > points[j].z$ or $points[i].x > points[j].x$. This suggests that when all points from indices $1...i$ are projected on the x-y plane, point i should be Non-Dominating if it is to be Non-Dominating in 3D, otherwise it is dominated by some existing point.

Since this step (already proven in the previous part) is correctly handled by the *checkForNonDominating* function, we can say that the point is Non-Dominating iff the call on the function *checkForNonDominating*

returns TRUE.
Hence, the points added to the list *nonDominatingPoints* are indeed Non-Dominating.

Time Complexity Analysis

The time complexity for sorting using Z Coordinates is $O(nlog(n))$.

As proven in the previous part, the time complexity of each the function *checkForNonDominating* is $O(log(i))$ and since $i < n$, every such operation is bounded by $O(log(n))$. Since the rest of the operations are only $O(1)$, we can say that the total time complexity is $O(nlog(n))/$

Question 2

We are given -

$$F_i = \sum_{j<i} \frac{Cq_jq_i}{(j-i)^2} - \sum_{j>i} \frac{Cq_jq_i}{(j-i)^2}$$
$$\therefore \frac{F_i}{q_i} = \sum_{j<i} \frac{Cq_j}{(j-i)^2} - \sum_{j>i} \frac{Cq_j}{(j-i)^2}$$

Assume two polynomials $p(x)$ and $q(x)$ -

$$p(x) = \sum_{1}^{n} Cq_ix^i$$
$$q(x) = \sum_{i=0,i!=n}^{2n} \frac{x^i}{(i-n)^2} * (-1 \text{ if i} < \text{n})$$

Let the product of these two polynomials be $S(x) = \sum S_ix^i$ -

for $0 < i <= n$

$$S_{n+i} = \sum_{j<i} \frac{Cq_j}{(j-i)^2} - \sum_{j>i} \frac{Cq_j}{(j-i)^2}$$
$$\implies S_{n+i} = \frac{F_i}{q_i}$$

Hence, from this, we can say that applying polynomial multiplication on $p(x)$ and $q(x)$, the coefficients will give us all the values we require. **(Polynomial Multiplication Discussed in Class)**

Time Complexity Analysis

Since the order of both the polynomials is $O(n)$, we can say that the multiplication will take $O(nlogn)$ time.