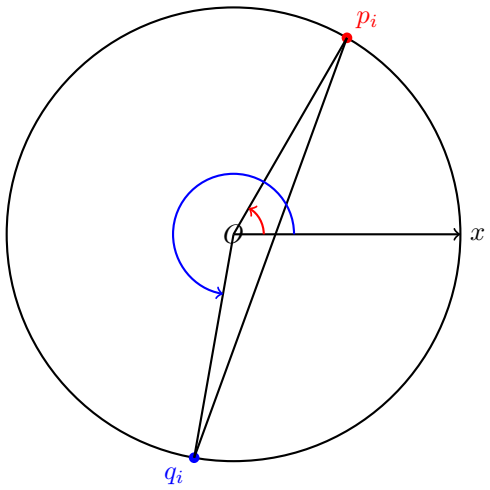
# CS345: Design and Analysis of Algorithms

Assignment 1

Submitted by: Aditi Khandelia, 220061

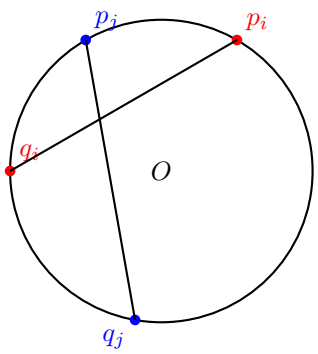
**Question 1.**
Given two sets $p_1, p_2, ..., p_n$ and $q_1, q_2, ..., q_n$ of n points on the unit circle, connect each point $p_i$ to the corresponding point $q_i$. Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect in $O(nlog^2n)$ time.

**Answer.**
**Approach.**
We will assume that $p_i$ and $q_i$ denote the counter-clockwise angular distance of the point from the line parallel to the x-axis made passing through the centre of the circle.
For every chord, we assume that $p_i < q_i$. If this is not the case, we will swap $p_i$ and $q_i$ to ensure uniformity across the input.

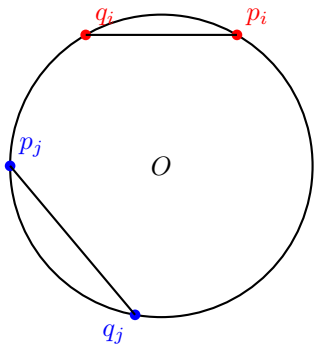



Next we arrange $p_i$ in ascending order, permuting $q_i$ appropriately to maintain correspondence with $p_i$. The following 3 configurations of $p_i, p_j, q_i, q_j$ are possible, when $i < j$, i.e. when $p_i < p_j$ :

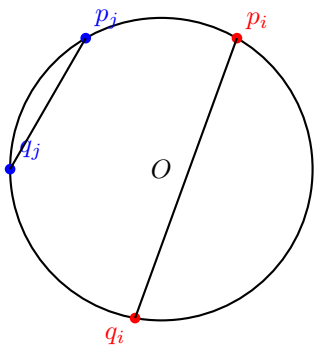
1. $p_i < p_j < q_i < q_j$:
   When this is the case, intersection of chords $i$ and $j$ takes place. We also notice that there is one inversion w.r.t. $i$ and $j$, wherein $p_j < q_i$.




2. $p_i < q_i < p_j < q_j$:
   When this is the case, no intersection of chords takes place. There are no inversions either.




3. $p_i < p_j < q_j < q_i$:
   When this is the case, no intersection of chords takes place. However, there are two inversions wherein $p_j < q_i$ and $q_j < q_i$.




Additionally, we notice that whenever there is an inversion in $q$, i.e. $q_j < q_i$, we have the third configuration and it contributes 2 inversions to the total no. of inversions.
Thus, let the no. of total inversions in the selected combined ordering of $p$ and $q$ be $invTotal$
Let the no. of inversions in the ordering of $q$ be $invQ$
Then, the total no of intersections $intersections$ will be :

$$intersections = invTotal - 2 * invQ$$

**Pseudocode.**
We now give the pseudocode for the aforementioned algorithm.

1. Iterate through the array $p$ and $q$ simultaneously, and swap $p_i$ and $q_i$ when $q_i < p_i$.
2. Sort $p$ and permute $q$ such that $pos_{p_i}$ in $p = pos_{q_i}$ in $q$.
3. Create an array $r$ by adding $p_i$, followed by $q_i$ to it, for $i = 1, 2, \ldots, n$.
   We now have an array of the following form :

$$p_1, q_1, p_2, q_2, \ldots, p_n, q_n$$

4. Calculate the no. of inversions in $r$. Let this value be $invTotal$.
5. Calculate the no. of inversions in $q$. Let this value be $invQ$.
6. Return $invTotal - 2 * invQ$

**Algorithm 1** Swap Elements in Arrays $p$ and $q$

```
1: function SwapElements(p[1..n], q[1..n])
2:     for i ← 1 to n do
3:         if q[i] < p[i] then
4:             Swap p[i] and q[i]
```

**Algorithm 2** Sort $p$ and Permute $q$

```
1: function SortAndPermute(p[1..n], q[1..n])
2:     Create array indices[1..n]
3:     for i ← 1 to n do
4:         indices[i] ← i
5:     Sort p and indices based on values in p
6:     Create new array q_new[1..n]
7:     for i ← 1 to n do
8:         q_new[i] ← q[indices[i]]
9:     q ← q_new
```

**Algorithm 3** Create Array $r$

```
1: function CreateArrayR(p[1..n], q[1..n])
2:     Initialize array r[1..2n]
3:     j ← 1
4:     for i ← 1 to n do
5:         r[j] ← p[i]
6:         r[j + 1] ← q[i]
7:         j ← j + 2
8:     return r
```

**Algorithm 4** Count Inversions in an Array

```
1: function CountInversions(q[1..n])
2:     if n = 1 then return 0
3:     mid ← ⌊n/2⌋
4:     left ← q[1 : mid]
5:     right ← q[mid + 1 : end]
6:     leftInv ← CountInversions(left)
7:     rightInv ← CountInversions(right)
8:     sort right
9:     splitInv ← 0
10:    for i ← 1 to mid do
11:        j ← BinarySearch(right, left[i])
12:        splitInv ← splitInv + j - 1
13:    return leftInv + rightInv + splitInv
```

**Algorithm 5** Main Pipeline

```
1: function MainPipeline(p[1..n], q[1..n])
2:     SwapElements(p, q)
3:     SortAndPermute(p, q)
4:     r ← CreateArrayR(p, q)
5:     invTotal ← CountInversions(r)
6:     invQ ← CountInversions(q)
7:     return invTotal − 2 ∗ invQ
```

**Time Complexity Analysis.**

1. The process of ensuring that $p_i < q_i \ \forall \ i$ takes $O(n)$ time since we need to traverse arrays $p$ and $q$ once, and swapping 2 elements takes $O(1)$ time.

2. Sorting $p$ takes $O(n \log n)$ time. For permuting, we traverse all values of $i$ and create a new array in $O(n)$ time. Copying $q_{new}$ to $q$ takes $O(n)$ time.

3. Creation of array $R$ takes $O(n)$ time since we traverse the values of $i$ from 1 to $n$ exactly once, copying elements in $O(1)$ time.

4. Counting inversions takes $O(nlog^2n)$ time :

   (a) The array is broken into two parts of equal size in $O(n)$ time and the no. of inversions in these is counter recursively. Let the time complexity of this be $T(n/2)$.

(b) Right subarray is sorted in $O(n \log n)$ time.

(c) For each element in left subarray, binary search is performed in $O(\log n)$ time, resulting in total time complexity $O(n \log n)$. Thus,

$$T(n) = O(n \log n) + T(n/2) * 2$$

By Master Theorem, $T(n) = O(nlog^2 n)$

Total time complexity $= O(n) + O(n \log n) + O(n) + O(nlog^2 n) = O(nlog^2 n)$

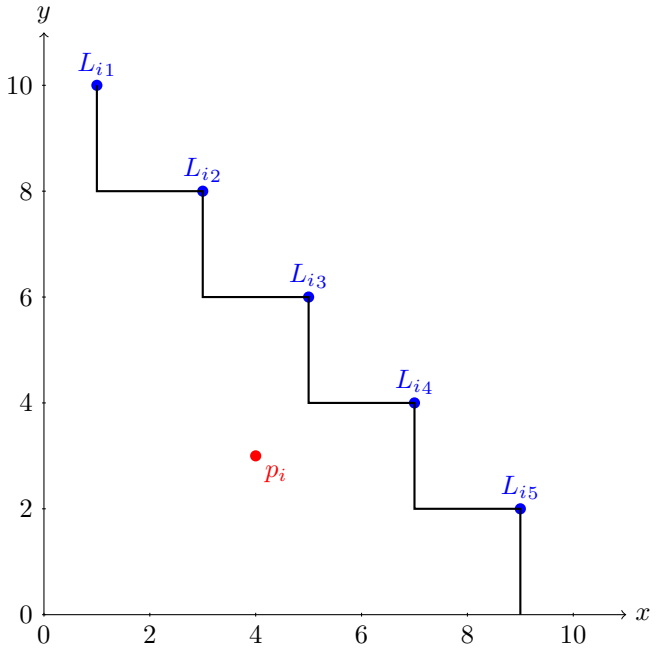
**Question 2.**
The concept of non-dominated points can be extended to three dimensions. However, applying a divide-and-conquer strategy to compute non-dominated points in 3D is not straightforward. One might consider reducing the problem to three instances of a 2-dimensional problem by projecting the points onto the xy, yz, and xz planes. However, this approach is incorrect (take some time to think about why). Interestingly, there exists a simple and elegant algorithm that uses a basic data structure to compute non-dominated points in 3D. The purpose of this exercise is to help you realize this fact.

**a.**
**Online Algorithm for 2D Non-Dominated Points :** Design an algorithm that receives n points in the xy-plane one by one and maintains the non-dominated points in an online manner. Upon insertion of the $i^{th}$ point, the algorithm should update the set of non-dominated points in $O(\log i)$ time. Note that it is acceptable if your algorithm guarantees a bound of $O(\log i)$ on the total time for the insertion of i points. It is not necessary for your algorithm to achieve an $O(\log i)$ bound on the processing time for the insertion of the $i^{th}$ point.

**Answer.**
**Approach.**
We will exploit the staircase structure of non-dominated points for the calculation.
Staircase structure implies that when the non-dominated points are arranged in the increasing order of x, they will be in decreasing order of y.

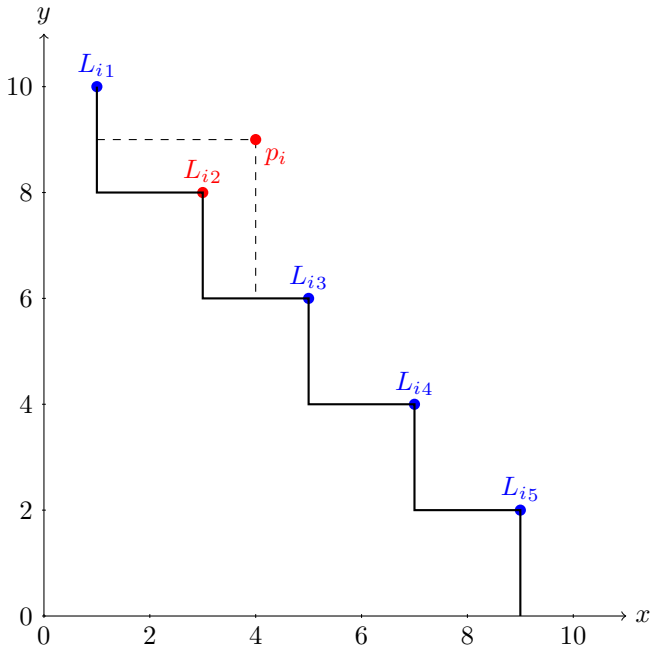
On receiving the $i^{th}$ point, we do the following:

1. Suppose that we have the list of non-dominated points received until this point in increasing order of $x$.
2. Using binary search, we can find the appropriate location for $i^{th}$ point in the list of non-dominated points, on the basis of its x-coordinate. Let this position be $j$ and the list of non-dominated points currently be represented by $L_i$. After that, the following two cases arise :
   (a) $y_{p_i} <= y_{L_{ij+1}}$ :
   In this case, the current point is not a non-dominated point. Consider the point $L_{ij+1}$. Since the non-dominated points are present in increasing order of $x$, $x_{p_i} <= x_{L_{ij+1}}$. Also, given $y_{p_i} <= y_{L_{ij+1}}$. Thus, this point is dominated by $L_{ij+1}$. Thus, we don't need to insert the current point in the list, and its order of increasing $x$ is maintained.




   (b) $y_{p_i} > y_{L_{ij+1}}$ :
   In this case, the point is surely a non-dominated point.
   Consider any point $L_{ik}$ where $k < j$. Since the points are arranged in increasing order of $x$, $L_{ik} <= x_{p_i}$, thus it is not dominated by any points on its left.
   Consider any point $L_{ik}$ where $k > j$. Since the points are arranged in increasing order of $x$ and are non-dominated, they are arranged in decreasing order of $y$ as well. Since $y_{p_i} > y_{L_{ij+1}}$, and $y_{L_{ij+1}} > y_{L_{ik}} \forall k >= j+1$. Hence $y_{p_i} > y_{L_{ik}}$ where $k > j$. Thus, the current point is not dominated by any point to its right.

Consider any received point that is a dominated point, i.e. is not present in the list. Let this point be $q$. $\exists\ L_{ik}$ such that $x_{L_{ik}} > x_q$ and $y_{L_{ik}} > y_q$. However, since none of the non-dominated points dominates the current point, either $y_{p_i} > y_{L_{ik}}$ or $x_{p_i} > x_{L_{ik}}$. WLOG, assume $y_{p_i} > y_{L_{ik}}$. Now, $y_{L_{ik}} > y_q$. Thus, $y_{p_i} > y_q$, which implied that no dominated point can dominate the current point.
Thus, the current point will be non-dominating.




3. In case we encounter the second case, we will insert the current point to its rightful position in the list of non-dominated points received yet. However, we will also have to delete certain points from the list since they might now be dominated by $p_i$ :
   (a) Consider any point $L_{ik}$, where $k < j$.
   These points have $x_{L_{ik}} < x_{p_i}$. In case $y_{L_{ik}} < y_{p_i}$, we will have to delete $L_{ik}$.
   (b) Consider any point $L_{ik}$, where $k > j$.
   Since these points have $x_{L_{ik}} > x_{p_i}$, they can't be dominated by $p_i$. Hence, we don't need to remove them.

On performing the given steps, we get the list of non-dominated points in increasing order of $x$ after every iteration.

**Pseudocode.**
Following is the pipeline :
We store the non-dominated points in a self-balancing binary search tree, on the basis of its $x$-coordinate.The BST is augmented with $y$ coordinate. On receiving the $i^{th}$ point, do the following :

1. Insert $p_i$ in the BST.
2. Calculate the successor of $p_i$. Let this be $x_{L_{ik}}$.
3. Compare $y_{p_i}$ and $y_{L_{ik}}$. If $y_{p_i} < y_{L_{ik}}$, delete $p_i$ from the BST and return.
4. Extract predecessor of $p_i$. Let this be *pred*.
   (a) If $y_{pred} > y_{p_i}$, return.
   (b) Else, delete *pred* from BST and go to 4.

The final BST will contain all the non-dominated points in the given set.

**Algorithm 6** Maintain Non-Dominated Points in BST

```
1: function ProcessPoint(p_i, BST)
2:     Insert p_i into BST
3:     L_ik ← Successor(BST, p_i)
4:     if y_i < y_{L_ik} then
5:         Delete p_i from BST
6:         return
7:     pred ← Predecessor(BST, p_i)
8:     while pred ≠ NULL do
9:         if y_pred > y_i then
10:            return
11:        else
12:            Delete pred from BST
13:            pred ← Predecessor(BST, p_i)
```

**Time Complexity Analysis.**
Consider the receipt of the $i^{th}$ point. Until this point, the BST can contain a maximum of $i$ points.

1. Since insertion in a BST takes $O(\log n)$ time, when the no. of insertions is $n$, the total time for insertion until this point is $O(i \log i)$.

2. The time complexity of calculating the predecessor is $\log n$ for a self-balancing BST like red-black tree. Time for removal of a node is also $O(\log n)$. For every point, the no. of time predecessor is calculated is one more than the no. of times deletion is undertaken.
   Lets consider the no. of deletions until this point. The maximum no. of deletions can be $O(n)$ since a point that is deleted is never inserted again. Thus, the no. of times predecessor is called is $O(n) + O(n) = O(n)$.
   The total time complexity for predecessor calculation : $O(n) * O(\log n) = O(n \log n)$
   Similarly, the total time for deletions : $O(n) * O(\log n) = O(n \log n)$

Thus, final time complexity for calculating the non-dominated points amongst a set of $i$ points :

$$O(i \log i) + O(i \log i) + O(i \log i) = O(i \log i)$$

It is worth noting that individual insertions needn't take $O(\log i)$ time since the no. of deletions in the $i^{th}$ iteration isn't determined, but the total time for $i$ iterations is $O(i \log i)$.

**b.**
**Algorithm for $3D$ Non-Dominated Points:** Design an $O(n \log n)$ time algorithm to compute the non-dominated points of a set of $n$ points in $3D$. You must carefully use the result from the previous part in your design.

**Answer.**
Answering this question requires careful application of the result from the previous question.
**Approach.**
We will convert the $3D$ problem into a $2D$ problem by imposing restrictions on accessing the third dimension. Let the point in the $i^{th}$ iteration be $p_i$.

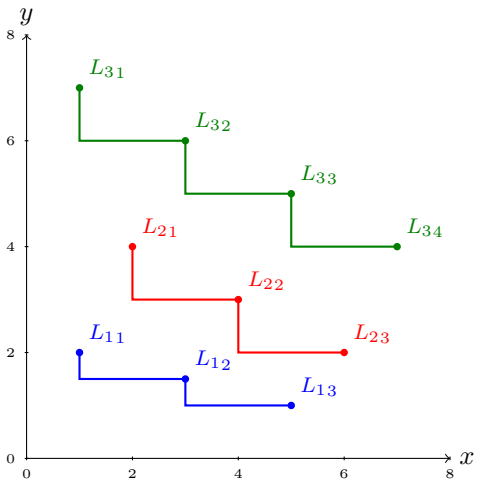
1. We will initially sort all the points in decreasing order of $z$. We will then process the points in online algorithm form, wherein we begin from the point with the highest $z$ coordinate, and build the set of dominated points for all the yet seen points. After processing the $n^{th}$ point, we get the actual list of non-dominated points.

2. The first point will definitely belong in the list of non-dominated points. This can be proven as follows :
   This point will have the highest $z$ coordinate since we are traversing the array in the decreasing order of $z$. Thus, for any point $p_j$ where $j \neq 1$, $z_{p_j} < z_{p_1}$. Hence, no $p_j$ can dominate $p_1$.
   We will thus insert the first point in the set of non-dominated points.

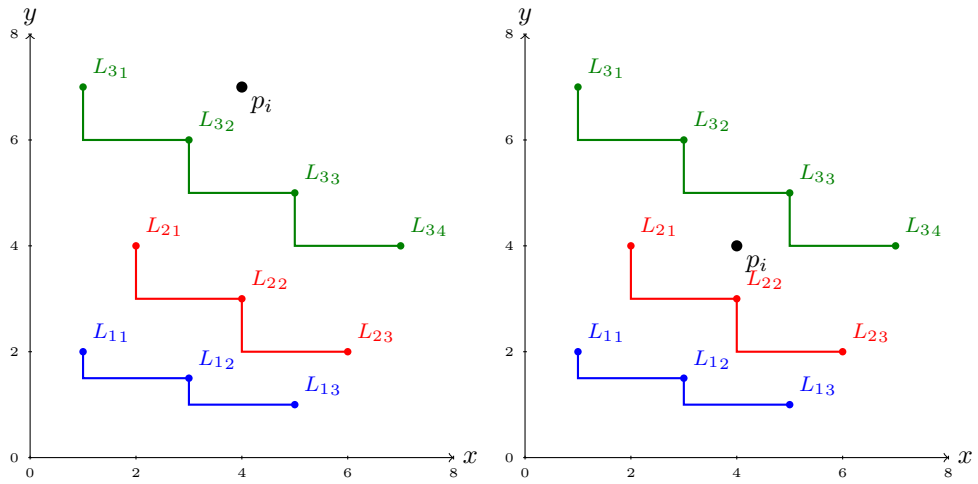
3. Consider the case of traversing the $i^{th}$ point:

   (a) This point can't dominate any of the points seen yet. This is because all of the points seen yet have higher $z$ than the current point. Thus, we don't need to delete any points from the set of non-dominated points.

   (b) We only need to decide if we should add the current point to the list of non-dominating points. Now, consider the $2D$ projections of the yet seen points.




The above diagram illustrates what the $2D$ projections will look like. Consider any point $L_{2j}$ and the corresponding $2D$ dominating point $L_{3k}$ as depicted in the diagram. $x_{L_{2_j}} < x_{L_{3_k}}$ and $y_{L_{2_j}} < y_{L_{3_k}}$. However, since both $L_{2j}$ and $L_{3k}$ are non-dominated in 3 dimensions, $z_{L_{2_j}} > z_{L_{3_k}}$. If this were not true, $y_{L_{2_j}}$ would be dominated by $y_{L_{3_k}}$. Now, consider the placement of $p_i$.




Consider the first configuration. Here the point is the part of the highest staircase.The highest staircase represents the non-dominating points in the $2D$ plane. We will prove that when $p_i$ is a part of the highest staircase, it is non-dominated by all points seen before. Since the point is a part of the highest staircase, there exists no other points $q$ such that $x_{p_i} < x_q$ and $y_{p_i} < y_q$. If this were the case, the point would lie below the highest staircase, i.e. be dominated by it in the $2D$ plane.

Consider the second configuration. Here the point lies below the highest staircase, i.e. is dominated by it in the $2D$ plane. Thus, there exists a point $q$ such that $x_q > x_{p_i}$ and $y_q > y_{p_i}$. Additionally, since the points are traversed in decreasing order of $z$, $z_q > z_{p_i}$. Thus, $q$ will dominate $p_i$.

(c) We thus only need to check if the current point is non-dominating in 2 dimensions, since we necessarily know that it is in the third dimension. For this, we will maintain two arrays : one to store the list of all dominating points seen yet, and the other to store the list of non-dominating points( out of the list of all dominating points) in the $2D$ plane.

**Pseudocode.**
We will have one self-balancing BST to store the set of non-dominated points in the $2D$ plane. We will also have a list that stores the list of all points that are non-dominated in 3 dimensions. Following is the pipeline :

1. Arrange points in decreasing order of $z$.
2. For the $i^{th}$ point, do the following:
   (a) Modify the ProcessPoint function from the previous function to modify the set of non-dominated points in case the current point belongs to it, or return false otherwise.

(b) In case the previous step returns true, add this point to the list of non-dominated points. Else, continue.

3. Return the list of all non-dominated points.

**Algorithm 7** Maintain Non-Dominated Points in 3D

```
function MaintainNonDominated3D(P)
    Sort P in decreasing order of z-coordinate
    Initialize empty BST and empty list ND
    for each point p_i(x_i, y_i, z_i) in P do
        if ProcessPoint2D(p_i, BST) = true then
            Add p_i to ND
    return ND
```

**Algorithm 8** Process Point in 2D BST

```
function ProcessPoint2D(p_i, BST)
    Insert p_i into BST
    L_ik ← Successor(BST, p_i)
    if y_i < y_{L_ik} then
        Delete p_i from BST
        return false
    pred ← Predecessor(BST, p_i)
    while pred ≠ NULL do
        if y_pred > y_i then
            return true
        else
            Delete pred from BST
            pred ← Predecessor(BST, p_i)
    return true
```

**Time Complexity Analysis.**

1. Sorting the points in decreasing order of $z$ coordinate takes $O(n \log n)$ time.

2. As proven in 2(a) above, the process of maintaining dominating points in $2D$ takes $O(n \log n)$ for n points, with $O(1)$ operation of adding a new point to the list of non-dominated points. Thus, final time complexity :

$$O(n \log n) + O(n \log n) + O(n) = O(n \log n)$$