ANALYSIS OF ALGORITHM
ASSIGNMENT 2

SUBHASREE JAYACHANDRAN (66222414), SREELALITHA BUDIGI (36717336)

1. LARGEST SUM PROBLEM

a) Formulate a divide-and-conquer algorithm for solving the Largest Sum Problem. Provide a description in words and pseudo code.

ALGORITHM:

Largest_Sum (int [] arr, int left, int right)

1. *Begin*
2. *If (left == right)*
   Return arr[left]//return the only element in array.
3. *Mid <- (left + right)/2*
4. *Left_Max_Sum <- least Integer value.*
5. *sum=0*
6. *For Loop I = mid to left //Decrementing i from mid to left*
   a. *Begin*
   b. *sum = sum + arr[i]*
   c. *If sum > Left_Max_Sum*
      Left_Max_Sum <- sum.
   d. *End FOR LOOP*
7. *Right_Max_Sum <- least Integer Value*
8. *Set sum = 0*
9. *For Loop I = mid + 1 to right + 1 //Iterating i from mid + 1 to right*
   a. *Begin*
   b. *sum = sum + arr[i]*
   c. *If sum > Right_Max_Sum*
      Right_Max_Sum <- sum
   d. *End FOR LOOP*
10. *max_Sum_LR <- max (Largest_Sum (arr, left, Mid), Largest_Sum (arr, Mid + 1, right))*
11. *Return max (max_Sum_LR, Left_Max_Sum + Right_Max_Sum)*
12. *End Program*

DESCRIPTION:

- Input : Array of integer of size n, Left Index, Right Index
- Output : Sum of the largest subarray
- Technique Used : Divide and Conquer Approach
- Time Complexity : O(nlogn)

Step 1: To check if the Left index is equal to the right index. If equal, then output the only element in the array as output. Go to Step 7. Otherwise, Go to Step 2.
Step 2: Set middle element to average of left index and right index. Using the middle element find the left subarray and right subarray.
Step 3: Compute the maximum subarray sum for left subarray in a recursive manner.
Step 4: Compute the maximum subarray sum for right subarray in a recursive manner.
Step 5: Computer the maximum subarray sum for a subarray with middle element.
Step 6: Output the maximum of the sums from step 3, step 4, and step 5.
Step 7: End

b) Give a mathematical proof of your algorithm's correctness.

Let arr be an array of numbers and we must prove Largest_Sum algorithm returns the maximum contiguous subarray sum in the given arr. We are going use proof by Induction for this.

Base Case:

1) If there are no elements in the arr then we return -1.
2) If there is only one element in arr then we return that element as our maximum contiguous subarray sum.
3) For number of elements in arr n > 1:

    - We assume inductively that for any given sub sequence of the arr whose size is less than n the Largest_Sum (sub sequence of arr) will return the maximum contiguous subarray sum of that sub sequence.
    - We get the max (maximum contiguous subarray sum of all sub sequences of arr) which will be max_sum_all_subsequence.
    - Now we finally find the max (max_sum_all_subsequence, total of maximum contiguous subarray sum of all sub sequences of arr) which will give us the maximum contiguous subarray for the entire arr.
    - Let us consider an arr of length n. We split this array into two equal halves left and right. We compute the maximum contiguous subarray for both the halves. Then we get the max (left_half_max, right_half_max) which will be max_left_right.
    - Now max {max_left_right, left_half _max + right_half_max} <= maximum contiguous subarray sum in the given arr

    We consider the following 3 possibilities to how maximum contiguous subarray lies with respect of left and right halves.

    1) If maximum contiguous subarray is in the left half, then from our inductive hypothesis Largest_Sum (arr, left, mid) = maximum contiguous subarray sum in the given arr which means
    max {max_left_right, left_half _max + right_half_max} >= left_half_max = maximum contiguous subarray sum in the given arr.

    2) If maximum contiguous subarray is in the right half, then from our inductive hypothesis Largest_Sum (arr, mid + 1, right) = maximum contiguous subarray sum in the given arr which means
    max {max_left_right, left_half _max + right_half_max} >= right_half_max = maximum contiguous subarray sum in the given arr.

    3) If maximum contiguous subarray starts in left index and ends in right index, then from our inductive hypothesis, then we will be finding the maximum contiguous subarray sum that starts in left index and ends in Right index.
    max {max_left_right, left_half _max + right_half_max} >= max_left_right = maximum contiguous subarray sum in the given arr.

    Thus max {max_left_right, left_half _max + right_half_max} = maximum contiguous subarray sum in the given arr.

c) State and mathematically prove the order Notation for the running time for your algorithm.

Largest_Sum () method is a recursive method. Therefore, the recurrence relation is represented as follows, T(n) represents the running time of the Largest_Sum () algorithm for an array of size n,

- Base case takes θ (1) when the left and right are same index, that is if the array is of size 1 then it returned as output sum -> *T (1) =1*
- Since we divide the array into two halves and find the maximum sum in both, it should take *T(n/2)* to solve one half. Hence to solve both halves it takes ->*2T(n/2)*
- To find the maximum sum that has the middle element we will use the full array of size n, hence this takes -> *θ(n).*

$$T(n) = \begin{cases} \theta(1) & ; n=1 \\ T(n/2) + T(n/2) + \theta(n) & ; \text{Otherwise} \end{cases}$$

which can also be written as,

$$T(n) = \begin{cases} 1 & ; n=1 \\ 2T(n/2) + n & ; \text{Otherwise} \end{cases}$$

The solution to the above recurrence relation is O(nlogn).

Mathematical proof for the order-notation is as follow:

From the recurrence relation mentioned above,

$T(n) = 2T(n/2) + n$ - [I]

Substituting *2T(n/4) + n/2 for T(n/2)* in the above equation, we get,

$=2(2T(n/4) + n/2) + n$

$=4T(n/4) + 2n$

Substituting *2T(n/8) + n/4 for T(n/4)* in the above equation we get,

$=4(2T(n/8) + n/4) + 2n$

$=8(T(n/8) + 3n$

$=2^3(T(n/2^3) + 3*n$

Going forward, for any positive integer k, T(n) satisfies the following recurrence relation,

$T(n) = (2^k) * T(n/2^k) + kn$ - [II]

Let us prove [ii] using the proof by induction method,

Step 1: To prove when k=1,

$T(n) = 2T(n/2) + 1(n)$

$T(n) = 2T(n/2) + n$

Thus from [i] when k=1, the recurrence relation is true.

Step 2: Assume that [ii] is true for k-1,

$T(n) = 2^{(k-1)} T(n/2^{(k-1)}) + (k-1) n$ - [III]

Step 3: To prove [i] for k,

From [iii],

$T(n) = 2^{(k-1)} T(n/2^{(k-1)}) + (k-1) n$

$T(n) = (2^{(k-1)}) * (2*T(n/k) + n/2^{(k-1)}) + (k-1) * n$

$T(n) = (2^k) * (T(n/2^k)) + n + (k-1) n$

$T(n) = (2^k) * T(n/2^k) + kn$

Therefore, [ii] is true.

Now Assume, $T(n/2^k) = T(1)$

$n/2^k = 1$

$n = 2^k$

$2^k = n$

$k = \log n$ -

Hence [ii], $T(n) = (2^k) * T(n/2^k) + kn$ becomes

$T(n) = nT(1) + n\log n$

$= n + n\log n$

In the above equation since n log n is the maximum degree it is the solution to the recurrence relation.

Therefore, $T(n) = O(n \log n)$

Thus Proved.

2. SEARCH FOR A TARGET NUMBER IN A ROW-WISE COLUMN-WISE SORTED M X N INTEGER MATRIX

a) PSEUDOCODE

int [] SearchTarget (int [] [] matrix, int rows, int columns, int target)

1. *Begin Program*
2. *For Loop i=0 to rows*
3. *Begin*
4. *int x=BinarySearch(matrix[i] ,0, columns, target)*
5. *if (x! =-1)*
6. *return [i, x]*
7. *End if*
8. *End for Loop*
9. *return [-1]*
10. *End Program*

int BinarySearch (int [] arr, int leftindex, int rightindex, int target)

1. *Begin*
2. *If rightindex >= leftindex*
    a. *mid=leftindex +(rightindex-leftindex)/2*
    b. *if arr[mid]==target*
        return mid.
    c. *else if arr[mid] > x*
        return BinarySearch (arr, leftindex, mid-1, target)
    d. *else*
        return BinarySearch (arr, mid+1, rightindex, target)
    e. *End If*
3. *End if*
4. *return -1 //target is not present in the array.*
5. *End BinarySearch*

Algorithm in words and Description:

Input: A m x n row-wise column-wise sorted matrix, target value, total no of rows, total no of columns.
Output: Index of the target element if the target is present. Otherwise [-1].
Algorithm Approach: Divide and Conquer

Step1: Start.
Step2: Input the Two-dimensional row-wise column-wise sorted matrix. Iterate the matrix row-wise. Perform Binary Search on every row.
Step3: If the Element is found, return the index of the element in the matrix. Other return [-1]
Step4: End.

SearchTarget:

- Input the Two-dimensional row-wise column-wise sorted matrix, no of rows, no of columns and target value.
- Iterate over all rows and perform binary search on each row.
- If element is found return the row and column number in which the element is present. Otherwise returns [-1].

BinarySearch:

Binary Search implements ***Divide and Conquer Approach*** to search for an element.

- Inputs are one-dimensional sorted array, left index, right index, and a target value.
- The middle index is computed as an average of left index and right index and the element in the middle index is compared to the target element.
- The middle index is returned if the target matches with the middle element.
- Else if targetvalue is greater than middle element, then the target value can only be in the right half starting after middle element. So, perform recursion on the right half.
- Else the targetvalue is smaller and So the left half is recurred.

The time complexity for a binary search is ***O(logn)***, where n is size of the array.
The total time complexity of the algorithm is ***O(nlogn)*** as we iterate n rows and perform Binary Search.

b) Give a mathematical proof of your algorithm's correctness.

We will be iterating each row and calling binary search to find the target element. We need to prove that BinarySearch returns the index of the target element if it is present in each row else it returns [-1].

Proof by Induction:

1. Let number of elements in each row be n which is given by.

    n = rightindex – leftindex + 1

Base case:

1. For n=0 the rightindex must be less than leftindex, so the algorithm returns [-1]. This is true as we cannot find the target in n empty rows)
2. For n=1 there is only one element so the mid will point that element and as per the algorithm if mid matches our target then we return the index else we return [-1].
3. For n>1 we have three cases that the algorithm checks for
    - Case 1: target value is greater than mid value.
      As it is a sorted matrix the target must be in right side of row from mid, so we search recursively on that half and return the index if found else return [-1].
    - Case 2: target value is lower than mid value.
      As it is a sorted matrix the target must be in left side of row from mid, so we search recursively on that half and return the index if found else return [-1].
    - Case 3: target value is equal to mid value.
      Return the index.

Always the (mid + 1 to rightindex) and (leftindex to mid – 1) < (0 to rightindex). So, the recursive approach will converge to a point where the case 3 is reached incase if the target element is present else will return [-1].

c) State and mathematically prove the order Notation for the running time for your algorithm.

Recurrence relation to search an element in a sorted matrix is as follows,

We perform iteration on all rows and perform binary search in every row. Therefore,

The Total time to find an element in a row-wise column-wise sorted matrix is.

= (Time take to iterate all the rows) * (Time taken to perform binary search on each row)

Time to iterate all the rows = m (No of rows)

Time to perform binary search is as follows,

T(n) represents the running time for BinarySearch on an array of size n.

Base case: When leftindex is greater than the rightindex, it takes O (1).

To divide the array into two halves, middle index is computed which is the average of the left index and right index, this takes O (1). Since the array is sorted the target can be less than or greater than the middle element so only half the array will be considered every time, therefore, time taken is half the length of array -> *T(n/2)*

T(n) = T(n/2) +O(C)

Hence the recurrence relation is as follows,

T(n) ={ *θ(1)* ;n<=1

T(n/2) + *θ (C)* ;n>1

Which is simplified to,

T(n) = { 1 ;n=1

T(n/2) + 1 ; n>1}

The above recurrence gives O (log n) time complexity.

<u>Mathematical Proof</u>:

The recurrence relation is as follows,

T(n) = T(n/2) +1

Substituting *T(n/2^2) +1 for T(n/2),* we get

T(n) = T(n/2^2) +1+1

=T(n/4) +2

Substituting *T(n/8) + 1 for T(n/4)* we get,

=T(n/8) +1+2

= T(n/8) +3

=T(n/2^3) +3

Going forward for any positive integer k, we get,

T(n) =T(n/2^k) +k

Assume, *n/2^k =1*

2^k = n

K =log n -

Using *k=log n*

T(n) = T (1) + log n

T(n) =1+ log n

T(n) =log n

In the above relation, since log n is having maximum degree for n, the order notation for the running time of binary search is O (log n).

Time taken to perform binary search on an array(row) of size n, O (log n)

Time taken to perform search on a row-wise and column-wise sorted matrix	=No of rows * time taken to perform BinarySearch on each row. = m log(n)

For a square matrix no of rows is equal to the no of columns, m=n

Therefore, the total time to find an element in a sorted square matrix is nlogn.

3. FIND CITY SKYLINES

a) ALGORITHM:

List<int []> getSilhouette (int [] [] Buildings):

1. *Begin Function*
2. *If (length (Buildings)==0)*
   Return Empty List
   Else:
   Return recursiveContourCalculation (Buildings,0, length (Buildings)-1)
3. *End Function*

LinkedList<int []> recursiveContourCalculation (int [] [] Buildings left, int right):

1. *Begin*
2. *If(left<right)*
   Mid<- left+(right-left)/2.
   Return mergeBuildings (recursiveContourCalculation (Buildings, left, Mid),
   recursiveContourCalculation (Buildings, Mid+1, right))
3. *Else*
   Result <- Empty LinkedList of One-Dimensional Array
   Result.add ({Buildings. [left][0], Buildings. [left][1]})
   Result.add ({Buildings. [left][2],0})
   Return Result
4. *End Function*

LinkedList<int []> mergeBuildings (Linkedlist<int []> leftSkyLine, Linkedlist<int []> rightSkyLine):

1. *Begin Function*
2. *Result <- Empty LinkedList of 1-d Array*
3. *h1=0, h2=0*
4. *while (length(leftSkyLine) > 0 and length(rightSkyLine)>0)*
   FinX=0, Finh=0
   if (left of First item of leftSkyLine < left of First item of rightSkyLine)
   FinX <- left of First item of leftSkyLine.
   h1<- height of First item of leftSkyLine
   Finh<- Max (h1, h2)
   Remove First item of leftSkyLine.

   Else if (left of first item of leftSkyLine > left of First item of rightSkyLine)
   FinX<- left of First item of rightSkyLine.
   h2 <- height of First item of rightSkyLine
   Finh <- Max (h1, h2)
   Remove First item of rightSkyLine.

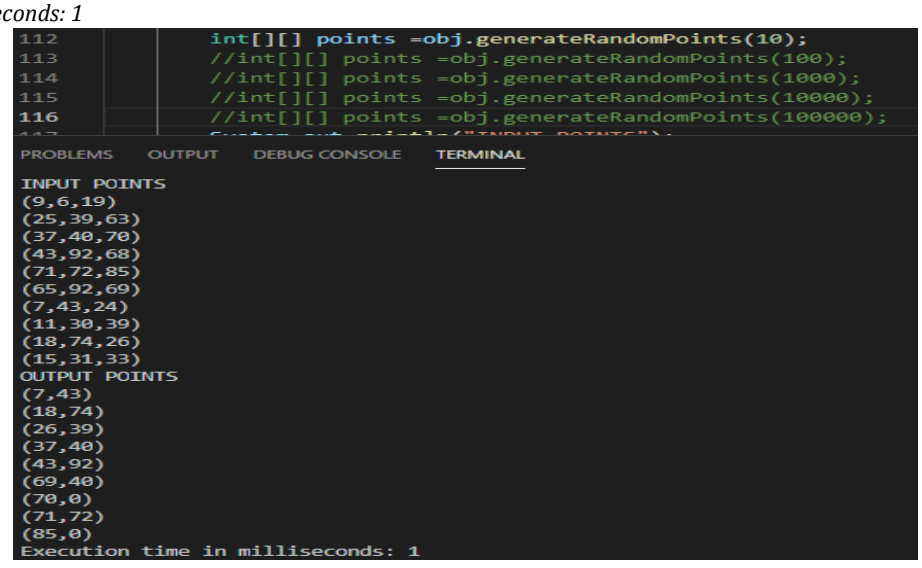
   Else
   FinX <-left of First item of leftSkyLine.
   h1<-left of First item of leftSkyLine.
   h2<-left of First item of RightSkyLine.
   Finh<- Max (h1, h2)
   Remove First item of leftSkyLine.
   Remove First item of rightSkyLine.
   EndIf
   If (Length (Result) ==0 or Finh is not equal to height of last item in Result)
   Result.add ({FinX, Finh})
   Result <- Append all items of leftSkyLine.
   Result<- Append all items of rightSkyLine.
   Return Result
5. *End Function*

DESCRIPTION:
Input: Array of Buildings, Buildings are represented by a 1-d Array of Left Index(left), Height(height), Right Index (right).
Output: A 2-d Array of Skyline of building.
Technique Used: Divide and Conquer Approach
Time Complexity: O(nlogn)

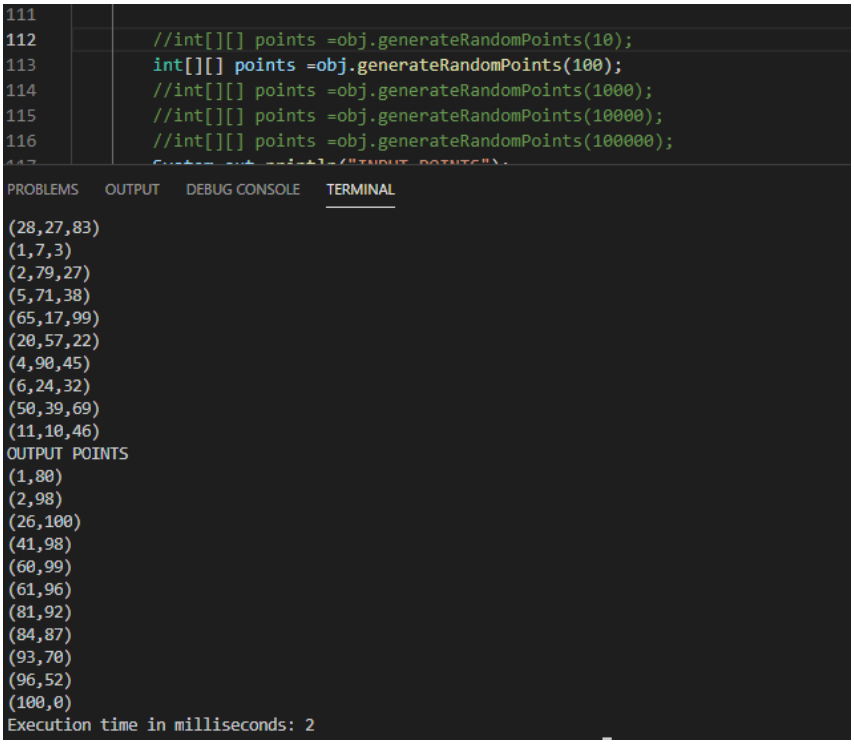
- Divide the Array of buildings into two halves, leftSkyline and RightSkyline.
- Compare the left of first item of both halves. Initialize x, h1 and h2 to 0.
- Update h1 if left of first item of LeftSkyLine is smaller, otherwise update h2 to height of first item of RightSkyline.
- Set h to the maximum of h1 and h2, and x to minimum left.
- Add [x, h] to the result.
- Remove the item with minimum left and Loop again till Length of either of the Two halves becomes 0.
- If left of first item of LeftSkyLine is equal to left of first item of RightSkyLine, x is set left of first item of LeftSkyLine and Both h1 and h2 are updated to the corresponding heights. Set h to maximum of h1 and h2.
- Remove both items form LeftSkyLine and RightSkyLine and Loop again till Length of either of the Two halves becomes 0.
- Add of remaining elements of the SkyLine to the Result if the other SkyLine empties.
- Output the two-dimensional Array containing Skyline for the buildings.

SAMPLE OUTPUT AND SCREENSHOTS:

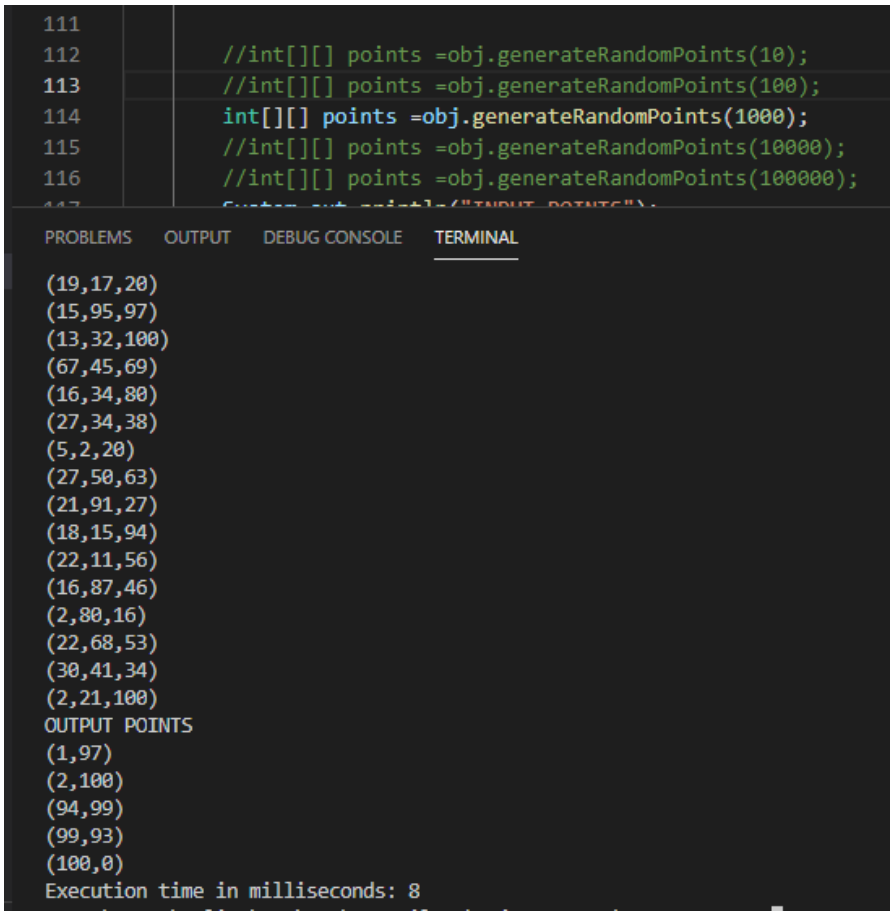
For 10 random points
INPUT POINTS
(9,6,19)
(25,39,63)
(37,40,70)
(43,92,68)
(71,72,85)
(65,92,69)
(7,43,24)
(11,30,39)
(18,74,26)
(15,31,33)
OUTPUT POINTS
(7,43)
(18,74)
(26,39)
(37,40)
(43,92)
(69,40)
(70,0)
(71,72)
(85,0)
Execution time in milliseconds: 1



For 100 random points



For 1000 random points



For 10000 random points

```
111
112         //int[][] points =obj.generateRandomPoints(10);
113         //int[][] points =obj.generateRandomPoints(100);
114         //int[][] points =obj.generateRandomPoints(1000);
115         int[][] points =obj.generateRandomPoints(10000);
116         //int[][] points =obj.generateRandomPoints(100000);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
(63,39,73)
(64,78,85)
(1,47,9)
(12,70,55)
(55,91,60)
(1,64,7)
(2,71,42)
(29,70,64)
(64,98,94)
(87,79,96)
(52,83,61)
(9,56,40)
(28,40,88)
(41,80,78)
(5,4,37)
(48,32,84)
(19,92,21)
(37,92,57)
(12,84,77)
OUTPUT POINTS
(1,100)
(100,0)
Execution time in milliseconds: 24
```

For 100000 random points

```
112         //int[][] points =obj.generateRandomPoints(10);
113         //int[][] points =obj.generateRandomPoints(100);
114         //int[][] points =obj.generateRandomPoints(1000);
115         //int[][] points =obj.generateRandomPoints(10000);
116         int[][] points =obj.generateRandomPoints(100000);
117         System.out.println("INPUT POINTS");
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
(96,34,99)
(50,70,100)
(75,19,79)
(15,80,18)
(34,20,83)
(18,51,75)
(27,51,76)
(12,22,17)
(7,60,61)
(5,20,31)
(11,11,15)
(17,41,25)
(31,11,86)
(38,25,85)
(10,9,19)
(1,84,2)
(12,94,72)
(46,23,94)
(25,39,26)
OUTPUT POINTS
(1,100)
(100,0)
Execution time in milliseconds: 121
```

Time Complexity Calculation:
N = 10
Our program execution time was 1 milli sec.

N = 100
Our program execution time was 2 milli sec.

N = 1000
Our program execution time was 8 milli sec.

N = 10000
Our program execution time was 24 milli sec.

N = 100000
Our program execution time was 121 milli sec.

We can see that the execution time is increasing as the numbers of data points are increased. Therefore, this shows that the algorithm follows an O (nlogn)