CSE 204: Data Structures and Algorithms

Assignment - 8: Divide and Conquer

Student Name: Zannatul Naim
Id: 1805024
Course No: CSE 204

Complexity analysis:

The recursive function is :

```
public static List<Distance> secondClosestPoint(List<Point> pointsX, int
start, int end)
```

Here `List<Point> pointsX` represents the points sorted by x
The cost is T(n)

The base cases:

```
        if (size == 2) {
            List<Distance> distances = new ArrayList<>();
            distances.add(new Distance(pointsX.get(start),
pointsX.get(end)));
            Point temp = pointsX.get(start);
            if (pointsX.get(end).y < pointsX.get(start).y) {
                pointsX.set(start, pointsX.get(end));
                pointsX.set(end, temp);
            }
            return distances;
        }
        if (size == 3) {
            List<Distance> distances = new ArrayList<>();
            distances.add(new Distance(pointsX.get(start),
pointsX.get(start + 1)));
            distances.add(new Distance(pointsX.get(start + 1),
pointsX.get(end)));
            distances.add(new Distance(pointsX.get(start),
pointsX.get(end)));
            Collections.sort(distances, (d1, d2) -> {
                double dist = d1.distance - d2.distance;
                if (dist < 0)
                    return -1;
                if (dist > 0)
                    return 1;
                return 0;
            });
```

```
            List<Point> temp = new ArrayList<>();
            temp.add(pointsX.get(start));
            temp.add(pointsX.get(start + 1));
            temp.add(pointsX.get(end));
            Collections.sort(temp, (d1, d2) -> {
                double dist = d1.y - d2.y;
                if (dist < 0)
                    return -1;
                if (dist > 0)
                    return 1;
                return 0;
            });
            for (int i = 0; i < 3; i++) {
                pointsX.set(start + i, temp.get(i));
            }
            if (distances.get(0).distance == distances.get(1).distance) {
                distances.remove(0);
            } else
                distances.remove(2);
            return distances;
        }
```

In the base cases, the cost is O(1)

Divide step:

```
    int mid = (start + end) / 2;
    List<Distance> distances1 = secondClosestPoint(pointsX, start, mid);
    List<Distance> distances2 = secondClosestPoint(pointsX, mid + 1,end);
```

The total cost of the divide step is **2T(n/2)**

Conquer step:

Step 1:

```
List<Distance> minDistance = new ArrayList<>(distances1);
        for (int i = 0; i < 2; i++) {
```

```
            if (distances1.size() >= 1) {
                if (distances1.get(0).distance ==
distances2.get(0).distance)
                    distances1.remove(0);
                if (distances2.size() == 2 && distances1.size() >= 1) {
                    if (distances1.get(0).distance ==
distances2.get(1).distance)
                        distances1.remove(0);
                }
            }
        }
        minDistance.addAll(distances2);
        Collections.sort(minDistance, (d1, d2) -> {
            double dist = d1.distance - d2.distance;
            if (dist < 0)
                return -1;
            if (dist > 0)
                return 1;
            return 0;
        });
        int minSize = minDistance.size();
        if (minSize == 4) {
            minDistance.remove(3);
            minDistance.remove(2);
        }
        if (minSize == 3)
            minDistance.remove(2);
```

The cost of step-1 is O(1)

Step 2:

```
merge(pointsX, start, mid, end);
```

Here's the merge() function,

```
private static void merge(List<Point> points, int low, int mid, int high)
{
        int i = low, j = mid + 1;
        for (int k = low; k <= high; k++)
            Main.temp[k - low] = points.get(k);
        for (int k = low; k <= high; k++) {
            Point temp1;
            if (i > mid)
```

```
                temp1 = Main.temp[j++ - low];
            else if (j > high)
                temp1 = Main.temp[i++ - low];
            else if (Main.temp[j - low].y < Main.temp[i - low].y)
                temp1 = Main.temp[j++ - low];
            else
                temp1 = Main.temp[i++ - low];
            points.set(k, temp1);
        }
    }
```

So, the cost of step -2 is O(n)

Step-3:

```
List<Point> midPoints = new ArrayList<>();

        for (int i = start; i <= end; i++) {
            if (pointsX.get(i).x >= pointsX.get(mid).x -
minDistance.get(0).distance
                    && pointsX.get(i).x <= pointsX.get(mid).x +
minDistance.get(0).distance) {
                midPoints.add(pointsX.get(i));
            }
        }
```

The cost of step-3 is O(n)

Step-4:

```
        int midPointSize = midPoints.size();

        for (int i = 0; i < midPointSize - 1; i++) {
            for (int j = 1; j <= 7; j++) {
                Distance midPointDistance;
                if (i + j < midPointSize) {

                    midPointDistance = new Distance(midPoints.get(i),
midPoints.get(i + j));
                    if (midPointDistance.distance !=
minDistance.get(0).distance
                            && midPointDistance.distance !=
minDistance.get(1).distance) {
                        minDistance.add(midPointDistance);
```

```
                    Collections.sort(minDistance, (d1, d2) -> {
                        double dist = d1.distance - d2.distance;
                        if (dist < 0)
                            return -1;
                        if (dist > 0)
                            return 1;
                        return 0;
                    });
                    minDistance.remove(2);
                }
            } else
                break;
        }
    }
    return minDistance;
```

The cost of step-4 is O(n)

So the total cost of conquer step is 3*O(n)+ O(1)= **O(n)**

So, **T(n)= 2*T(n/2) + O(n)**

Acording to Master theorem, **T(n)=a*T(n/b) + f(n)**
Here, a=2,b=2,f(n)=O(n)
Now, $\log_2 2=1$
So, f(n) is polynomially equal and follows the 2nd condition of Master's Theorem.
So, T(n)=Theta(n^1*logn)=Theta(n*logn)