Bangladesh University of Engineering and Technology

Department of Computer Science and Engineering

Academic Year 2020 - 2021

CSE 204
-Data Structures and Algorithms Sessional-

Offline No. 8
Study of Divide and Conquer Algorithms

Name: Anwarul Bashir Shuaib

Roll: 1805010

Section: A1

Date of Submission: June 18, 2021

Study of Divide and Conquer Algorithms

Abstract

In this experiment, we study a famous computational geometry problem of finding the second closest pair of points in a given set of two-dimensional points, using the Divide and Conquer approach. We analyze the asymptotic time complexity of our approach, mainly the part where we merge the solution of our two smaller sub-problems to get back the answer to our original sub-problem. We also show how this approach outperforms the traditional brute-force approach by a better asymptotic time complexity, as well as in terms of algorithmic ingenuity.

The Divide and Conquer Approach

The Divide and Conquer paradigm mainly consists of the following three steps:

- **Divide** the original problem into smaller sub-problems
- **Conquer** the sub-problems recursively
- **Combine** the solutions of the sub-problems into one for the original problem.

Here, we go through each step of our algorithms to solve the closest pair problem and analyze their respective complexity. But in order to make sure that our algorithms runs in $\mathcal{O}(n \log n)$, we need to do some pre-processing step.

Pre-processing Step

We first copy our given set of points to two arrays and sort them according to their x coordinate and y coordinate. Since we will be using MergeSort to accomplish this task, which has a time complexity of $\Theta(n \log n)$, we are still within our pre-specified time complexity. We do this in the following lines of our code:

```
8   public ClosestPair(ArrayList<Point> points) {
9       this.points = points;
10      sortedX = new ArrayList<>(points);
11      sortedY = new ArrayList<>(points);
12      sortedX.sort(Point.sortByX());
13      sortedY.sort(Point.sortByY());
14  }
```

Pre-processing time complexity: $\mathcal{O}(n \log n)$

The Base Cases

As we divide our original problems to smaller sub-problems, we are destined to reach the boundary where we can safely brute-force through the problems. We do this in our base case handling inside our recursive `findClosestPair` method.

```
31      if (sortedX.size() == 1)
32          return new PairOfPoints(null, null, Integer.MAX_VALUE);
33      else if (sortedX.size() == 2)
34          return new PairOfPoints(sortedX.get(0), sortedX.get(1),
                Point.sqrDist(sortedX.get(0), sortedX.get(1)));
35      else if (sortedX.size() == 3)
36          return closestAmongstThree(sortedX);
```

Here, line `36` simply calls a brute-force method to find the closest pair amongst three points.

Base case time complexity: $\mathcal{O}(1)$

The Divide Step

We create a recursive method `findClosestPair` which takes the two sorted versions of the array as input and and splits the arrays in two equal halves. Each half contains points sorted by monotonically increasing x coordinate. We do the same to the array which contains points sorted by monotonically increasing y coordinate.

```
37      int mid = sortedX.size() / 2;
38      ArrayList<Point> leftSortedX = new ArrayList<>(sortedX.subList(0, mid));
39      ArrayList<Point> leftSortedY = new ArrayList<>(sortedY.subList(0, mid));
40      ArrayList<Point> rightSortedX = new ArrayList<>(sortedX.subList(mid,
            sortedX.size()));
41      ArrayList<Point> rightSortedY = new ArrayList<>(sortedY.subList(mid,
            sortedY.size()));
```

We note that the splitting sub-routine takes at most $\mathcal{O}(n)$ time internal to the built-in Java data-structure.

Divide step time complexity: $\mathcal{O}(n)$

The Conquer Step

Here, we recursively pass the splitted and sorted versions of our original arrays to our `findClosestPair` method, and get back the closest pair on the left half and the right half of our input points region.

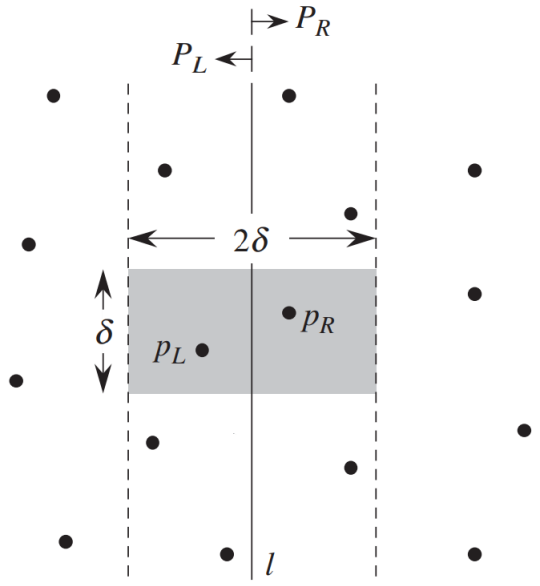
```
42      PairOfPoints leftBest = findClosestPair(leftSortedX, leftSortedY);
43      PairOfPoints rightBest = findClosestPair(rightSortedX, rightSortedY);
```

Here, line `42` and line `43` recursively calls our `findClosestPair` method. If we denote the total run-time of our original problem to be $T(n)$, then,

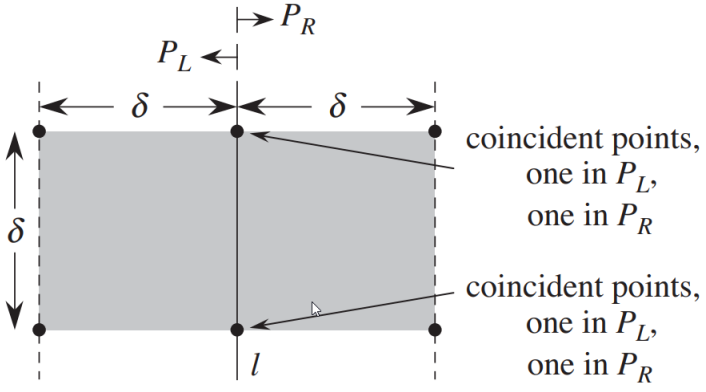
Conquer step run-time: $2T\left(\frac{n}{2}\right)$

The Combine Step

We focus our attention mainly to this step, because all of the algorithmic ingenuity that separates our approach from the brute-force method lies inside this part. We first pick our best candidate from the left and right part of the region. Let $\delta = min(leftBest.dist, rightBest.dist)$. We observe that if there happens to be a better candidate than what we have achieved so far, this candidate has to be consisting of one point from the left part, and one point from the right part, with atmost δ distance apart from the mid-point.



Now, we limit our searches to points within this strip, and it might first seem like we have to do $\mathcal{O}(n^2)$ searches to find the minimum distance among these points. This is the part where it gets most interesting. We note that both the points p_L and p_R must be at most 2δ apart from each other horizontally, and at most δ distance apart vertically. So, p_L and p_R must lie within a $\delta \times 2\delta$ rectangle, centered at the midpoint.



We also note that at most 8 points among all the points within the strip can lie within this $\delta \times 2\delta$ rectangle in the worst case. This is obvious from the figure above, since this is the worst possible arrangement the points can have. If we were to include one more point in this region, it

has to lie within a distance $\leq \delta$ from one of the points. So we can limit our search to at most 7 points following each points in our strip, and guarantee an $\mathcal{O}(n)$ performance of this step.

```
44        PairOfPoints best = (leftBest.dist <= rightBest.dist ? leftBest : rightBest);
45        PairOfPoints bestFromSplit = findSplitPair(sortedX, sortedY, best);
46        if (best.dist <= bestFromSplit.dist) return best;
47        return bestFromSplit;
```

And inside the `findSplitPair` method,

```
50    private PairOfPoints findSplitPair(ArrayList<Point> sortedX, ArrayList<Point>
          sortedY, PairOfPoints best) {
51        int midX = sortedX.get(sortedX.size() / 2).x;
52        int delta = best.dist;
53        ArrayList<Point> strip = new ArrayList<>();
54        for (Point p : sortedY) {
55            if (p.x >= (midX - delta) && p.x <= (midX + delta)) strip.add(p);
56        }
57        PairOfPoints splitBest = new PairOfPoints(null, null, Integer.MAX_VALUE);
58        for (int i = 0; i < strip.size(); i++) {
59            for (int j = i + 1; j <= Integer.min(i + 7, strip.size() - 1); j++) {
60                Point a = strip.get(i);
61                Point b = strip.get(j);
62                int dist = Point.sqrDist(a, b);
63                if (dist < splitBest.dist) {
64                    splitBest.dist = dist;
65                    splitBest.p1 = a;
66                    splitBest.p2 = b;
67                }
68            }
69        }
70        return splitBest;
71    }
```

Combine step time complexity: $\mathcal{O}(n)$

Second Nearest Point

The above algorithm finds us the nearest pair of points. To find the second closest pair, we just had to run the algorithm two more time. The pseudo-code explaining this step is as follows:

$cp \leftarrow findClosestPair(sortedX, sortedY)$
$a \leftarrow findClosestPair(sortedX - cp.p1, sortedY - cp.p1)$
$b \leftarrow findClosestPair(sortedX - cp.p2, sortedY - cp.p2)$
return $min(a, b)$

Summary

We conclude our total run-time by the combination of each individual steps from previous:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + \mathcal{O}(n) + \mathcal{O}(1) \\ &= 2T\left(\frac{n}{2}\right) + \mathcal{O}(n) \end{aligned}$$

In a nutshell,

$$T(n) = \begin{cases} \mathcal{O}(1) & n \leq 3 \\ 2T(\frac{n}{2}) + \mathcal{O}(n) & n > 3 \end{cases}$$

Comparing with the master theorem, $T(n) = aT(\frac{n}{b}) + f(n)$, we realize that

$$a = 2, b = 2, f(n) = \Theta(n)$$

$$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$$