

Demetri Karras

Kettering University

CS-203: Computing and Algorithms III

Professor Giuseppe Turini

June 12th, 2025

Algorithm Description

Points are randomly generated and placed into two lists. One list is sorted by x-coordinate in non-decreasing order, and the other by y-coordinate in non-decreasing order. These two lists are passed into the recursive method `getClosestPair` which contains the main logic of the algorithm.

The base case occurs when the number of examined points is less than or equal to 3. The closest pair is found using brute force for groups of 2 and 3 points. All point combinations are tested, and their distances are compared to find the closest pair.

The recursive case occurs when the number of examined points is greater than 3. The points are split by a vertical line that effectively divides the plane in half. The lists of points are divided into sublists representing the left and right halves of the plane, and these new sublists are passed into the method recursively to find the closest pair in both halves. The closer pair of both halves is found. The vertical strip surrounding the dividing line is examined for a possible closer pair made up of points from both halves. Points from the y-sorted list are placed into a new list if they are within the current closest pair's distance from the dividing line. Point combinations within this list are examined for a closer pair. Mathematical relationships are used to make this search efficient, checking the fewest possible points.

My code closely follows the algorithm described in the appendix. There are no deviations in my implementation, but I chose to create the following classes to increase the readability and logical breakdown of the code: `CoordinatePlane`, `CoordinatePlane.Point`, `CoordinatePlane.Solution`.

Theoretical Analysis

Input Size: number of points on the coordinate plane (n)

Basic Operations in relevant methods from CoordinatePlane.java

- `CoordinatePlane(int numPoints, int bounds)` - calls `buildPlane`
 - arithmetic, condition evaluation, assignment - can be considered one basic operation
 - $C(n) = 1$
 - calls `buildPlane`
 - $C(n) = n \log(n) + 2n$
 - $C(n) = n \log(n) + 2n + 1$
- `buildPlane()`
 - assignment, adding to a set - in a while-loop
 - calls sort from `java.util.List` (TimSort algorithm - using most common case)
 - $C(n) = n \log(n)$
 - calls `addAll` from `java.util.List`
 - $C(n) = n$
 - $C(n) = \sum_{i=0}^{n-1} (1) + n \log(n) + n = n \log(n) + 2n$
 - There exists an extremely unlikely worst case scenario where duplicate points are continuously generated forever, but only the most-common case will be used in this analysis
- `findClosestPairByBruteForce(List<Point> pointsSortedNonDecrByX)` - calls `getDistance(Point p1, Point p2)`
 - all operations can be considered one basic operation - no loops
 - arithmetic, assignment, element access, basic operations from called methods
 - $C(n) = 1$
- `getDistance(Point p1, Point p2)`
 - all operations can be considered one basic operation - no loops
 - arithmetic, assignment
 - $C(n) = 1$

* The main logic of the algorithm is in the `findClosestPair` method of `CoordinatePlane.java`, so the main analysis will occur on this method

Best Case

The best-case scenario occurs when the points are distributed in such a way that points near the dividing line never need to be examined.

```
// stores all points within the current minimum distance of the dividing line
s = q.stream()
    .filter(point -> Math.abs(point.x - midX) < minDistance)
    .toList();
```

This statement will result in an empty list `s`, preventing the for-loop with a nested while-loop from ever running.

Recursive Definition

Base Case: number of points ≤ 3

- Brute force is used to find the closest pair

Recursive Case: number of points > 3

- The input lists are split into left and right halves, and the closest pair in these halves is recursively found

Basic Operations

- `findClosestPairByBruteForce(List<Point> pointsSortedNonDecrByX)` is the method called for the base case
 - $C(n) = 1$
- 2 recursive calls
- `stream()` is called on lists 3 times - can be considered one linear operation
 - $C(n) = n$
- arithmetic, assignment, element access, conditional evaluation, and all other constant time operations can be considered as one basic operation - not in loops
 - $C(n) = 1$

Closed Form Formula

$$\begin{aligned}C(n) &= 2 * C(n/2) + (n + 1) \\&= 2 * (2 * C(n/4) + (n/2 + 1)) + (n + 1) \\&= 2 * (2 * (2 * C(n/8) + (n/4 + 1)) + (n/2 + 1)) + (n + 1) \\&= 2 * (4 * C(n/8) + (n/2 + 2) + (n/2 + 1)) + (n + 1) \\&= 8 * C(n/8) + n + 4 + n + 2 + n + 1 = 8 * C(n/8) + 3n + 7 \\&= 2^k C(n/2^k) + kn + (2^k - 1)\end{aligned}$$

Base Case: $n/2^k \leq 3 \rightarrow k \leq \log_2(n/3)$ *can be simplified* $\rightarrow k = \log_2(n) \rightarrow n = 2^k$

$= nC(\text{base}) + n\log_2(n) + (n - 1)$, base occurs when $n/2^k < 3$

$$= n\log_2(n) + 2n - 1$$

$$C(n) \in \Theta(n\log(n))$$

Master Theorem Application

1. $C(n) = 2 * C(n/2) + (n + 1)$
2. $a = 2, b = 2$
3. $f(n) = n + 1$
4. $f(n) \in O(n) \rightarrow d = 1$
5. $a = b^d \rightarrow 2 = 2^1$
6. $C(n) \in \Theta(n\log(n))$

Worst Case

The worst-case scenario occurs when all points are clustered around the line that divides the plane.

```
// stores all points within the current minimum distance of the dividing line
s = q.stream()
    .filter(point -> Math.abs(point.x - midX) < minDistance)
    .toList();
```

This statement will result in list s containing the same n number of points that were input into the problem. The for-loop will run $n - 2$ times, with the inner while-loop running no more than 5 times according to the appendix.

Recursive Definition

Base Case: number of points ≤ 3

- Brute force is used to find the closest pair

Recursive Case: number of points > 3

- The input lists are split into left and right halves, and the closest pair in these halves is recursively found
- A vertical strip is created near the dividing line based on the current minimum distance, and points within it are checked for a closer distance

Basic Operations

- `findClosestPairByBruteForce(List<Point> pointsSortedNonDecrByX)` is the method called for the base case
 - $C(n) = 1$
- 2 recursive calls
- `stream()` is called on lists 3 times - can be considered one linear operation
 - $C(n) = n$
- arithmetic, assignment, element access, conditional evaluation, and all other constant time methods and operations can be considered as one basic operation - not in loops
 - $C(n) = 1$
- the operations inside the for-loop in the vertical strip search around the dividing line can be considered constant (including the while-loop) based on the properties of the problem
 - $C(n) = \sum_{i=0}^{n-3} (1) = n - 2$

Closed Form Formula

$$\begin{aligned} C(n) &= 2 * C(n/2) + (n + 1 + (n - 2)) = 2 * C(n/2) + (2n - 1) \\ &= 2 * (2 * C(n/4) + (2(n/2) - 1)) + (2n - 1) \\ &= 2 * (2 * (2 * C(n/8) + (2(n/4) - 1)) + (2(n/2) - 1)) + (2n - 1) \\ &= 2 * (4 * C(n/8) + (n - 2) + (2(n/2) - 1)) + (2n - 1) \\ &= 8 * C(n/8) + (2n - 4) + (2n - 2) + (2n - 1) = 8 * C(n/8) + 6n - 7 \\ &= 2^k C(n/2^k) + 2kn - (2^k - 1) \end{aligned}$$

Base Case: $n/2^k \leq 3 \rightarrow k \leq \log_2(n/3)$ *can be simplified* $\rightarrow k = \log_2(n) \rightarrow n = 2^k$

$= nC(\text{base}) + 2n\log_2(n) - (n - 1)$, base occurs when $n/2^k < 3$

$$= 2n\log_2(n) + 1$$

$$C(n) \in \Theta(n\log(n))$$

Master Theorem Application

1. $C(n) = 2 * C(n/2) + (2n - 1)$
2. $a = 2, b = 2$
3. $f(n) = 2n - 1$
4. $f(n) \in O(n) \rightarrow d = 1$
5. $a = b^d \rightarrow 2 = 2^1$
6. $C(n) \in \Theta(n\log(n))$

Most Common Case

Because both the best case and the worst case have the same efficiency class, the most common case will share that same efficiency class.

$$\Omega(n\log(n)), O(n\log(n))$$

$$\Rightarrow \Theta(n\log(n))$$

****Important Note**


In all cases of this algorithm, constructing the plane and sorting the points has $n\log(n) + 2n + 1$ basic operations, falling into an efficiency class of $\Theta(n\log(n))$. This does not affect the efficiency class of the algorithm as a whole, and was omitted from the analysis for simplicity, but is still worth mentioning.

Graph

Title: Theoretical Time Efficiency

X-axis: input size

Y-axis: time

1  $y = x \log x$ 

2



Empirical Analysis

Hardware and Software

- CPU: 13th Gen Intel(R) Core(TM) i7-1355U 1.70 GHz
- RAM: 32 GB
- Operating System: Windows 11 64-bit
- Java Version: Java 23
- Execution Tool: IntelliJ IDEA 2025.1.1.1

Description

To conduct this analysis, I ran 7 trials for the following 6 distinct input sizes: [100, 1000, 5000, 10000, 20000, 50000]. I chose these values because they are varied and provide data that is easy to graph, with a visible trend. To ensure the experiment is kept consistent, I ensured that 20% of the total gridspace was occupied by points. This means that $[\text{numPoints} * 5 = ((2 * \text{bounds}) + 1)^2]$ must be true. I used Java's System.nanoTime method to time the algorithm's execution, and I converted the time into milliseconds. For each input size, the first trial was always an outlier that took much longer than the rest, so I omitted the "real" first trial.

Table

Time: milliseconds

Number of Points (n)	100	1,000	5,000	10,000	20,000	50,000
Bounds	11	35	79	111	158	250
Trial 1 Time	1	10	33	55	109	297
Trial 2 Time	1	6	20	53	113	239
Trial 3 Time	1	7	27	63	91	270
Trial 4 Time	1	6	25	58	116	302
Trial 5 Time	1	6	20	52	113	248
Trial 6 Time	1	9	26	65	98	292
Trial 7 Time	1	8	29	51	128	244
Average	1	7	25	56	109	270

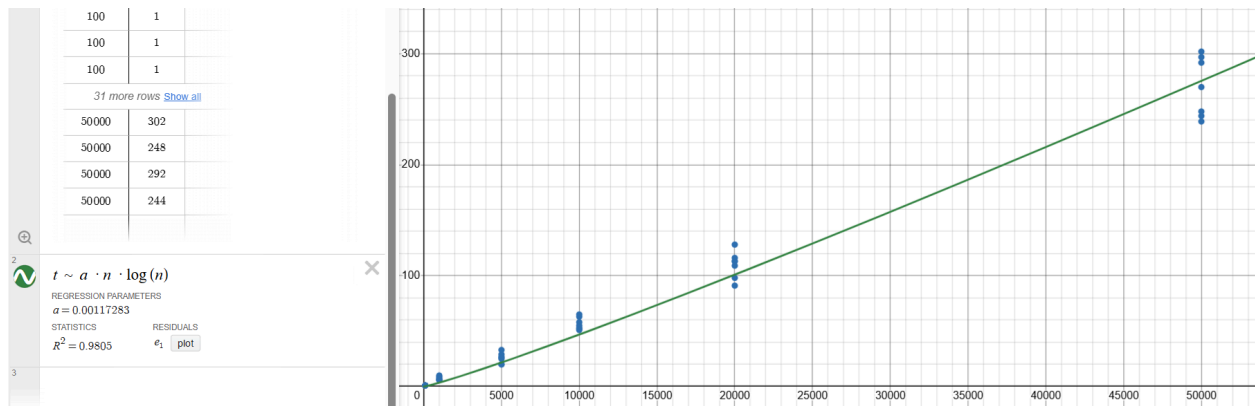
Graph

Title: Empirical Time Efficiency

X-axis: input size

Y-axis: time

Approximation Function: $0.00012n \log(n)$



Analysis Comparison

The Theoretical Analysis concluded with classifying this algorithm as $\Theta(n \log(n))$ time efficiency. There were distinct worst case and best case scenarios, but both led to the same efficiency class. The Empirical Analysis yielded results that complemented the Theoretical Analysis. The plotted data corresponds to a trendline of $n \log(n)$. Ultimately, both forms of analysis provide ample evidence that prove the efficiency class to be **$\Theta(n \log(n))$** .