# Lab 1 Analysis

EN.605.621
Foundations of Algorithms
Spring 2023

**Jose Márquez Jaramillo**

Engineering and Applied Science Programs for Professionals
Johns Hopkins University
United States
March 14th, 2023

## Contents

# 1 Algorithm construction - Closest Pairs

In an attempt to challenge my understanding of the material covered in the course, I provide two approaches to this problem. Two approaches are pursued:

1. A brute force approach where a nested loop is used to compute all of the pair-wise distances and merge sort is used to sort by distance and return the $m \leq \binom{n}{2}$ pairs with the lowest distance.

2. An adaptation of the classical implementation of the closest pair problem as presented in [1] section 5.4, [2] section 33.4, and [3] section 3.4. Despite my better efforts, I was able to achieve a correct solution although not an efficient one.

In order to preserve the level of effort across the entire assignment, I will provide an analysis of both algorithms rather than focusing on the trivial version. To begin, we will assume that no two points have the same $x$ coordinate or the same $y$ coordinate.

## 1.1 A Point class

In order to proceed with the calculations, I created a $Point$ class which can be found in the point.py file. Each point instance has x and y coordinates as its characteristics. It also has a couple of methods intended to return the value of the x and y coordinates denominated $Point.get()$ and finally a printing operator overload to print the value of the coordinate pair in a cartesian way, i.e. (x,y).

## 1.2 A brute force approach

The brute force approach corresponds to the quadratic time solution which uses an exhaustive search. That is, by computing the distance between each of the $\Theta(n^2)$ pairs of points one by one and returning the $m$ closest of them. The pseudo-code is shown next:

**Algorithm 1** Distance($p1$,$p2$)

**Inputs:** Two points $p1$ and $p2$
**Output:** The euclidean distance between $p1$ and $p2$

1: **return** $\sqrt{(p1_x - p2_x)^2 + (p1_y - p2_y)^2}$

**Algorithm 2** Brute-Force($P$,$m$)

**Inputs:** List $P$ of $n$ points and an integer value $m$ of closest pairs to be found
**Output:** A dictionary called $results$

1: $n \leftarrow$ length of points
2: Initialize a $distances$ dictionary
3: **for** $i = 1, 2, \ldots, n$ **do**
4: **for** $j = i + i \ldots, n$ **do**
5: $distances$[distance(points[$i$],points[$j$])] = (points[$i$], points[$j$])
6: **end for**
7: **end for**
8: Gather all of $distances$ keys into a list called $sorteddistances$
9: Sort $sorteddistances$ by use of mergesort()
10: Initialize a $results$ dictionary
11: **for** $i = 1, 2, \ldots, m$ **do**
12: Initialize a dictionary in $results[i]$
13: $results[i]$[distance] $\leftarrow sorteddistances[i]$
14: $results[i]$[points] $\leftarrow distances[sorteddistances[i]]$
15: **end for**
16: **return** $results$

As previously mentioned, lines 3-7 compute and store all $n^2$ distances. This indicates a worst-time complexity of $O(n^2)$. In line 8 we make use of mergesort, the particular implementation of mergesort is a modified version of the one proposed by [4] in section 6.11. This implementation can be found in the file mergesort.py along with the code. This implementation as discussed in class and in the previous texts mentioned is $O(n \log_2 n)$. Finally, lines 11-14 run in $O(m)$. In total we get

$$O(n^2) + O(n \log_2 n) + O(m) = O(n^2) \tag{1}$$

## 1.3 An attempt to divide and conquer

As previously mentioned, I tried to make use of a modified version of the classic example produced in the aforementioned texts of finding the closes single pair among a group of points. For reasons I will discuss later, I was not able to produce an algorithm that produced significant improvements over the Brute Force method. Very complete explanations have been written about this method, see for example section 5.4 of [1] for an in-depth explanation. Below I will produce the pseudo-code and explain where I think my code is not producing the desired divide-and-conquer efficiency gains. However, before getting into that, let's explore why finding a single pair of points work so well under the procedure provided in the textbooks mentioned:

In the case of finding the single closest pair of points, one first approach could be to consider the one-dimensional (say x-coordinates only) case. In the one-dimensional version, the problem can be solved using a single invocation of a sorting subroutine (say mergesort) followed by a single pass over the sorted points. When we are considering points in a plane (with x and y coordinates) the first obvious approach would be to sort by x-coordinates and by y-coordinates [3]. This is where we can make use of a pre-processing step. That is, we sort the inputs using the mergesort mentioned before ([4] in section 6.11) and sort the input points, once by x-coordinates and again by y-coordinates. This creates two copies of the point sets. The high level-plan is then analogous to the one that worked well for the other divide-and-conquer algorithms studied in class (including mergesort):

1. Split the input points into a ”left half” and a ”right half”, and
2. Look separately at subsets of size $m$ points in the left half (with a recursive call) and do the same on the right side.

There are significant geometric implications to the algorithm proposed in the sources mentioned beforehand. These geometric and mathematical implications I found to be most important when trying to find the single closest pair of points. In my implementation, some of the main ideas of divide and conquer are implemented although the perfect time complexity of $n \log n$ is not perfectly achieved for reasons that I will get into after presenting the pseudo-code. Having said that, the time complexity of the algorithm is $O(n \log n)$ just like other divide and conquer approaches of a similar division into halves. This will be shown graphically further along the report.

**Algorithm 3** DC-Closest-M-Pairs($P$,$m$)

**Inputs:** List $P$ of $n$ points and an integer value $m$ of closest pairs to be found
**Output:** A dictionary called $results$

1: $X \leftarrow P$ sorted by $x$-coordinate
2: $Y \leftarrow P$ sorted by $y$-coordinate
3: **return** DC-Closest-M-Pairs-Helper($X$,$Y$)

**Algorithm 4** DC-Closest-M-Pairs-Helper($X$,$Y$,$m$)

**Inputs:** Lists $X$,$Y$ and $m$, **Output:** A dictionary called $FinalResult$

**if** size of $X \leq m + 3$ **then**
    Brute-Force($X$,$m$)
**end if**
$\bar{x} \leftarrow$ median $x$-coordinate of $X$
$X_l \leftarrow$ points of $X$ to the left of $\bar{x}$
$X_r \leftarrow$ points of $X$ to the right of $\bar{x}$
$Y_l \leftarrow$ points in $Y$ which are also in $X_l$
$Y_r \leftarrow$ points in $Y$ which are also in $X_r$
$result_l \leftarrow$ DC-Closest-M-Pairs-Helper($X_l$,$Y_l$,$m$) ▷ Find the closest points in the left half
$result_r \leftarrow$ DC-Closest-M-Pairs-Helper($X_r$,$Y_r$,$m$) ▷ Find the closest points in the right half
$\delta_l \leftarrow$ Distance of the $m$th pair in $result_l$
$\delta_r \leftarrow$ Distance of the $m$th pair in $result_r$
$\delta \leftarrow \min\{\delta_l, \delta_r\}$
$Y' \leftarrow$ those points in $Y$ whose $x$-coordinate is within $\delta$ of $\bar{x}$
$result_f \leftarrow$ Initiate an empty list for distances of points between $Y'$
**for** $i \in$ length$(Y')$ **do**
    **for** $j \in i + 1 \min\{i + 7,$ length$(Y')\}$ **do**
        $p, q \leftarrow Y'[i], Y'[j]$
        $distance \leftarrow$ Distance$(p, q)$
        **if** $distance < \delta$ **then**
            $result_f[i] \leftarrow$ initiate dictionary instance of possible result
            $result_f[i]['distance'] \leftarrow distance$
            $result_f[i]['points'] \leftarrow p, q$
            $\delta \leftarrow distance$
        **end if**
    **end for**
**end for**
$OverallResult \leftarrow$ Initiate an empty dictionary for results
$FinalDistances \leftarrow$ Initiate an empty list for results
**for** $result \in [result_l, result_r, result_c]$ **do**
    **for** $i \in result$ **do**
        **if** $result[i]['distance'] \notin OverallResult$ **then**
            $OverallResult[result[i]['distance']] \leftarrow$ initiate entry
            $OverallResult[result[i]['distance']]['points'] \leftarrow result[i]['points']$
            Append $result[i]['distance']$ to $FinalDistances$
        **end if**
    **end for**
**end for**
MergeSort($FinalDistances$)
$FinalResult \leftarrow$ Initiate a final dictionary to return
**for** $i \in range(m)$ **do**
    $FinalResult[i] \leftarrow$ initiate a dictionary instance
    $FinalResult[i]['distance'] \leftarrow FinalDistances[i]$
    $FinalResult[i]['points'] \leftarrow OveralResult[FinalDistances[i]]['points]$
**end for**
**return** $FinalResult$

Now that we have given motivation for the algorithm and referenced the sources that do a complete explanation of its mathematical and geometrical correctness. Let's discuss its worst-time complexity. We can begin by inferring that this algorithm has a similar time complexity to that of the mergesort, although it is important to investigate further the loops that occur in the conquer step. If the conquer step runs in $O(n)$ then we can indicate that the overall algorithm runs in $O(n \lg n)$. Lines $1-3$ provide the base case, where the algorithm defaults to Brute-Force whenever the number of items in the current solution is $m+3$. We previously indicated that Brute-Force is $O(n^2)$, however, this instance gets only called once the algorithm is conquered and therefore this does not run in quadratic time with respect to $n$ although it can run quadratic with respect to $m$ and it is $O(n)$. Lines $4-10$ provide the divide step, since the list of points is evenly divided, we can assume that this step runs in $O(\lg n)$. The loop in lines $16-27$ runs in quadratic time with respect to the length of $Y'$, yet keep in mind that $Y'$ corresponds to a rather small number of points with the characteristic of having an $x$-coordinate within $\delta$ distance from $\bar{x}$. This means that although not linear with respect to the size of $Y'$, it is linear with respect to $n$ or $O(n)$. Next, we have the loop in lines $30-38$ which runs at most in quadratic time with respect to $m$, not with respect to $n$. We have the condition that $m \leq \binom{n}{2}$ which means that $3m^2 \leq n^2$. Therefore this step is also $O(n)$. Finally, the loop in lines $41-45$ runs linearly with respect to $m$. Since we have a dividing step that splits each time in half linearly with respect to $n$, we have a $O(n \lg n)$ algorithm. This will be graphically displayed in the next section.

# 2 Timing the algorithms

## 2.1 Brute Force vs. Divide and Conquer

Embedded into the code the grader will find that aside from the implementation of the algorithm, I have added counters for the operations which pertain to the algorithm. The MergeSort steps that happen during pre-processing along with the Merge-Sort steps that happen inside of the algorithm are not counted. This is because they are themselves $O(n \lg n)$ and do not affect the asymptotically worst-time complexity of each.

Within the Analysis folder of the programming assignment, you will find a jupyter-notebook called Analysis.jpny that generates that runs the code as well as generated the graphs below. Since Brute-Force is quadratic, it begins to slow down significantly for runs that include over 3,000 points. For benchmarking I ran both the Brute-Force method and the Divide and Conquer method for $m = 5$ for $100 \leq n \leq 3000$. I also added to the graph the corresponding values for $n^2$ and $n \lg n$:

### 2.1.1 Number of Operations

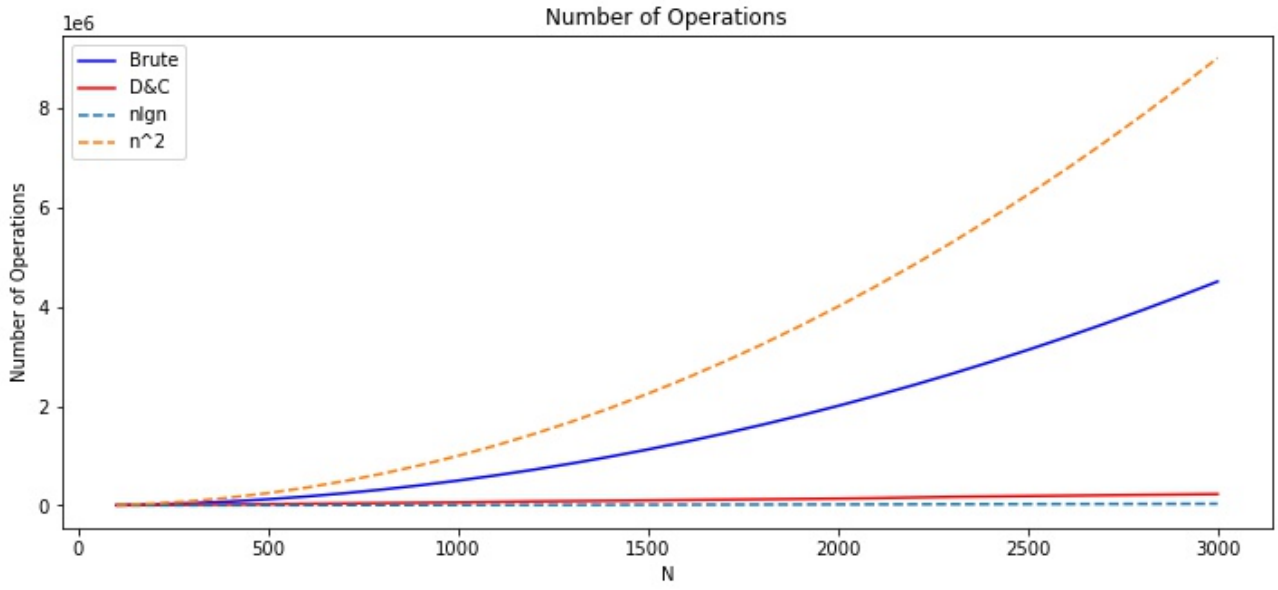


Figure 1: Number of Operations

The graph shows that the Brute-Force algorithm does run in $O(n^2)$ and the Divide and Conquer algorithm runs in $O(n \lg n)$.

### 2.1.2 Time of execution

I also measured the time it took to run the algorithms for the same parameters, the next graph shows how much faster the Divide and Conquer algorithm runs with respect to the Brute-Force algorithm.

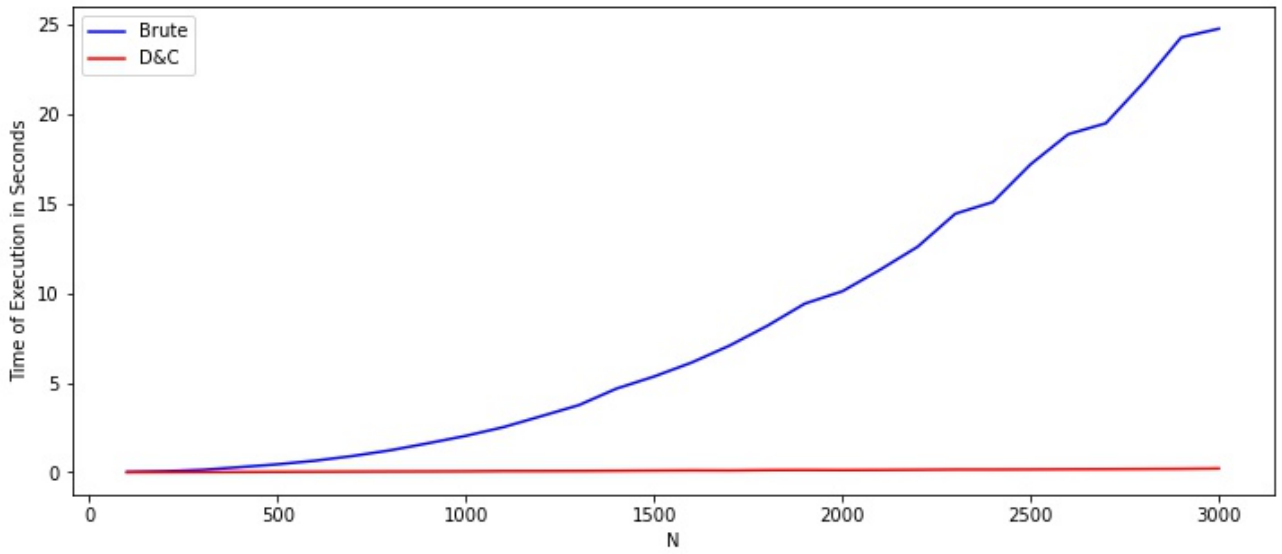


Figure 2: Execution Time

## 2.2 Divide and Conquer at higher levels of $n$

Given that we have shown how much more inefficient the Brute-Force algorithm is with respect to the Divide and Conquer algorithm, we will now turn emphasis to higher levels of $n$ values. I completed runs for $m$ and $10000 \leq n \leq 500000$. Below are the graphical representations of the number of operations with respect to $n \lg n$ as well as the execution time for the different levels of $n$.

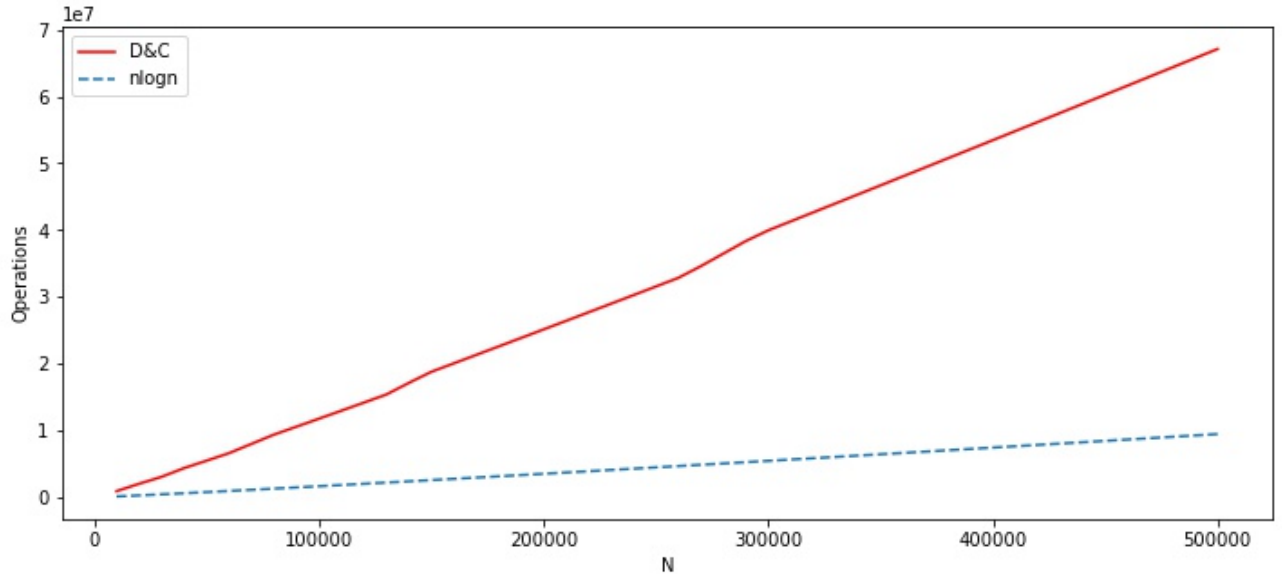


Figure 3: Execution Time

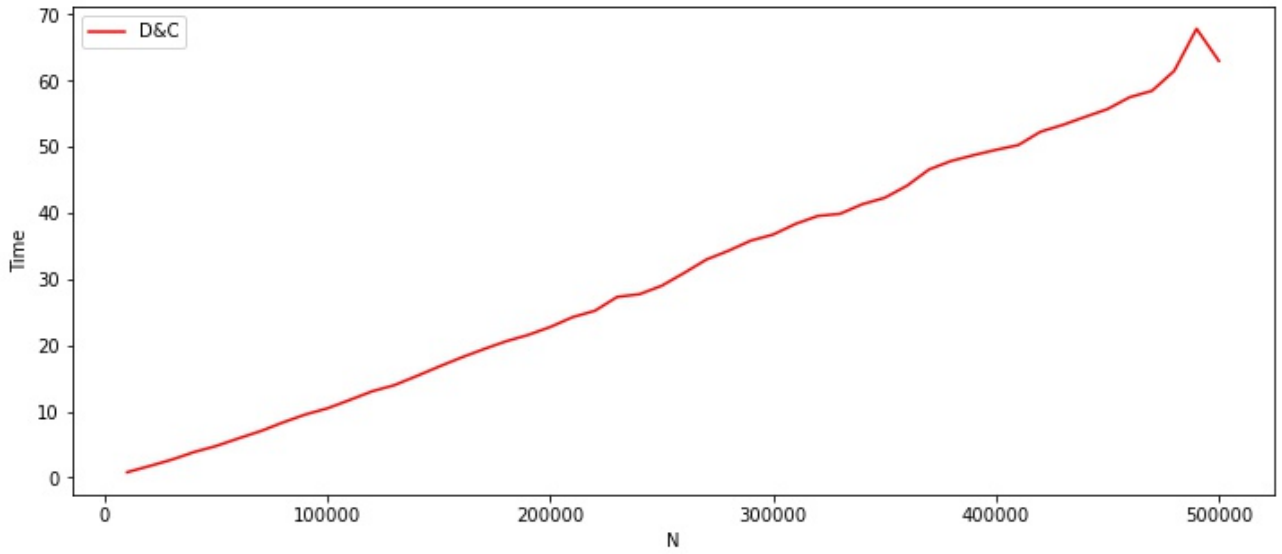


Figure 4: Execution Time

# 3 Retrospection and closing remarks

## 3.1 Retrospection

I have to begin by stating that I found finding a Divide and Conquer solution for this problem rather difficult. In my efforts to write the program, I had to review the material we were given in class several times along with reviewing the other materials cited previously. Since the assignment did not indicate a particular method of solving the problem, I was very close to giving up and not pursuing the divide-and-conquer method. Having endured I have to say that I have learned a lot about the divide-and-conquer method. And although I have still quite a bit to practice before I can intuitively write original divide-and-conquer algorithms, I am in a much better position than when I started the process.

With respect to possible improvements to the execution of the code, I believe that Algorithm 4 can be possibly improved during the conquer step. The loop in lines $31 - 38$ could possibly be omitted if there is mathematical proof that no repeated pairs are produced among $result_l, result_r$, and $result_f$. The MergeSorts used for pre-processing and for sorting during Algorithm 4 and Algorithm 1 can be changed by other more efficient sorting methods such as natural MergeSort.

## 3.2 File structure of the programming assignment

With respect to how the files are structured in the programming assignment Zip Folder, you will find this analysis along with the readme.md file and 4 other folders:

1. The Analysis/ folder contains the jupyter notebook and graphs used in this report to indicate run times and time complexity visually.
2. Finally, the SourceCode/ folder contains the python files used along with the Inputs/ and Outputs/ folders:
   (a) The Inputs/ folder contains sample input runs generated during the program. The set of points is generated using numpy's random module [5].
   (b) The Output/ folder contains sample output runs generated for testing of the program. Each time the main.py file runs one text file is generated for the Brute-Force and Divide-And-Conquer methods respectively.
   (c) main.py runs the code and calls upon the other files.
   (d) closespoints.py implements the algorithms mentioned in this report.
   (e) mergesort.py implements two different mergesorts: one for a python list, and one for a list of points of class Point().
   (f) point.py implements the Point() class as described in this report.

# References

[1] J. Kleinberg and □. Tardos, *Algorithm Design*. Addison Wesley, 2005.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 2009.

[3] T. Roughgarden, *Algorithms Illuminated : Part 1 : The Basics*. Soundlikeyourself Publishing, LLC.

[4] B. N. Miller and D. L. Ranum, *Problem Solving with Algorithms and Data Structures Using Python SECOND EDITION*. Franklin Beedle Associates.

[5] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2