**Kordel France**
**Lab 0**
**Class 605.621 Section 83**

# Lab 0 Analysis Document

*FranceLab0* compares different algorithms in their ability to find the closest pair of points in a group of 2-dimensional points on a Cartesian coordinate system. *FranceLab0* is a software module written in Python 3.7 that facilitates mathematical expressions to find such solutions.

## Brute-Force Algorithm

A *brute-force* algorithm is an algorithm that checks every possible combination of solutions in order to find the correct solution, usually without any heuristics. The algorithm developed as a brute-force method in finding the closest pair of points within a series of points is as follows:

---

```
function bruteForceAlgorithm( dataPointArray ) returns Float{
    minDistance = inf
    for index in dataPointArray {
        for subindex in dataPointArray {
            firstPoint = dataPointArray [index]
            secondPoint = dataPointArray [subIndex]
            distance = getDistanceBetweenPairs( firstPoint, secondPoint )
            if distance < minDistance {
                minDistance = distance
            } endif
        } endloop
    } endloop
    return minDistance
} endfunction
```

---

Effectively, the method above finds the distance between every possible combination of points within the dataset. The algorithm contains two simple *for-loops*, one nested within the other. Because of these loops, the worst-case asymptotic runtime for this algorithm is $O(n^2)$. Since there is no heuristic or optimizer used in this algorithm, the best-case and average-case runtimes are also on the order of $n^2$.

## Efficient Algorithm – Divide-and-Conquer Method

The "efficient" approach to finding the closest pair of points in a series of points is outlined below in pseudocode. It facilitates a "divide-and-conquer" approach similar to merge sort and binary search. I looked at utilizing a genetic algorithm as the more efficient algorithm but, while way more flexible, the runtime of the genetic algorithm far exceeded that of the brute-force method. Note that this is pseudocode for the *divide-and-conquer* algorithm and that the actual method in Python is slightly different.

---

```
function divideAndConquerAlgorithm( dataPointArray ) returns Float {
    minDistance = inf
    dataPointArray.sort( key = y )
    shortestDistance = findClosestDistance( dataPointArray, minDistance )
    return shortestDistance
} endfunction


function findClosestDistance( dataPointArray, minDistance ) returns Float {
    midpoint = dataPointArray.count / 2
    leftSubArray = dataPointArray[ 0 : midpoint ]
    rightSubArray = dataPointArray [ midpoint : dataPointArray.count ]
    deltaLeft = findClosestDistance ( leftSubArray, minDistance )
    deltaRight = findClosestDistance ( rightSubArray, minDistance )
    deltaMin = min ( deltaLeft, deltaRight )
    lowerBound = midpoint – deltaMin
    upperBound = midpoint + deltaMin
```

```
    stripArray = [ ]
    for index in leftSubArray {
        if ( leftSubArray[index]  > lowerBound ) & ( leftSubArray [index] < midpoint {
            stripArray.append ( leftSubArray [ index ]
        } endif
    } endloop

    for index in rightSubArray {
        if (rightSubArray[index]  < upperBound) & (rightSubArray[index]) > midpoint {
            stripArray.append ( rightSubArray [ index ]
        } endif
    } endloop

    stripDistance = findClosestDistance ( stripArray, minDistance )
    if stripDistance < minDistance {
        return stripDistance
    } else {
        return minDistance
    } endif
} endfunction
```

---

1. To begin, we approach the problem with a similar mindset of binary search. In order for binary search to be effective, the data must be sorted beforehand. In a like manner, our algorithm must first obtain sorted input data in order for it to be effective. We will see why in just a moment. We sort the data in ascending order according to the *y-coordinate* of the point via Quicksort.

   **Cost: *O ( n lg n )***

2. After the input data is sorted, we have a data array we will call the *dataPointArray.* We then find the midpoint of the array such that the midpoint is *dataPointArray.count / 2.* In the event, that the number of points within the array is odd, we round down to the nearest integer as the midpoint index. The midpoint now allows us to separate the array into two equal (or near equal) halves. At each point within the array.

   **Cost: *O ( n / 2 )***

3. Each time the array is split, we find the distance between all of the points within that smaller array. Let's denote the minimum distance of the points in the left and right subarrays as *deltaLeft* and *deltaRight* respectively. On the first split, the minimum distance value *deltaMin* is established as *min( deltaLeft, deltaRight ).* We've established an upper bound on how large the distance between two points can be with the result of *deltaMin.*

   **Cost: *O ( n )***

4. Next, we reconsider the midpoint of the array. We've found the minimum distance between all points within a subset of the dataset (in this case half of the dataset), but we need to find the minimum distance between points *across* datasets. To do so, imagine that the midpoint of *dataPointArray* divides all components within the array in 2-dimensional space with an imaginary vertical line. We know that a point within the left subarray and a point within the right subarray can *at most* be *deltaMin* distance apart across this vertical line. If we split *deltaMin* in half and attribute one half of its value to form the lower bound of a strip on the left side of the midpoint line, and the other half of its value to establish the upper bound of the strip on the right side of the line, we can evaluate all of the points that can possibly be smaller than *deltaMin* between both array halves. Consider the diagram below for reference:

   **Cost: *O ( n / 2 )***

5. We build another list and call it *stripArray* that contains all of the datapoints that fall between the lower and upper bound about the midpoint line described above. We evaluate the distance between all points contained within *stripArray* and find the minimum value. Denote this value as *stripDistance* and compare it to a global variable of *minDistance.* The minimum of the two becomes the new global value of *minDistance.*

   **Cost: *O ( n )***

6. Steps 2 – 5 define a subroutine that is performed recursively until the subarrays can no longer be split in half. Each time step 5 is completed, *minDistance* is compared with the

global value of *minDistance* and, if the former is smaller than the latter, the global *minDistance* is reset to the new low. The resulting global variable of *minDistance* represents the smallest distance between two pairs of points within the dataset *dataPointArray.*

**Cost: *O ( n lg n )***

Consolidating the costs from the steps above, we receive the following total asymptotic cost for our "efficient" divide-and-conquer algorithm:

$T(n) = O(n \lg n) + O(n / 2) + O(n) + O(n / 2) + O(n) + O(n \lg n)$

$= 2 \{ O(n \lg n) + O(n / 2) + O(n) \}$

$= O(n \lg n) + O(n) + 2O(n)$

$= O(n \lg n) + 3O(n)$

➔ $O(n \lg n) + O(n)$

# Cost Analysis for Both Algorithms

Please refer to the table below for acquired runtime costs for both algorithms in the *FranceLab0* module.

| *A* | *B* | | *C* | | *D* | |
|---|---|---|---|---|---|---|
| **n** | **Brute-Force** | | **Divide-And-Conquer (Not Including Sorting)** | | **Divide-And-Conquer (Including Sorting)** | |
| | expected | received | expected | received | expected | received |
| **6** | 36 | 36 | 15 | 12 | 46 | 43 |
| **10** | 100 | 100 | 33 | 22 | 99 | 88 |
| **25** | 625 | 625 | 116 | 57 | 348 | 289 |
| **100** | 10,000 | 10,000 | 664 | 256 | 1,992 | 1,584 |
| **200** | 40,000 | 40,000 | 1,528 | 469 | 4,585 | 3,526 |
| **200** | 40,000 | 40,000 | 1,528 | 465 | 4,585 | 3,522 |
| **1000** | 1,000,000 | 1,000,000 | 9,965 | 1,888 | 29,896 | 21,819 |
| **1000** | 1,000,000 | 1,000,000 | 9,965 | 1,979 | 29,896 | 21,910 |

*Table 1 - Cost values for the Brute-Force and Divide-and-Conquer Algorithms*

Columns B, C, and D indicate the three evaluations. For each of the evaluations in those columns, there values for the expected value of the cost and the actual received cost from the algorithm at each respective level of *n.* It's important to note that, since trace runs for *n > 100,*

are pseudo-randomly generated, the received values for the *divide-and-conquer* algorithm will be slightly different than what is shown in the above table when run on a different machine.

I originally only considered the number of calls to the *get_distance_between_pairs( )* function in *algorithm.py* as the metric to evaluate the cost of each algorithm. However, the divide-and-conquer algorithm requires the input array to be sorted in order to perform. As a result, there are two points in the algorithm where merge-sort is used to sort the array of points. The brute-force algorithm does not require the point array to be sorted so to only consider the number of calls to the *get_distance_between_pairs( )* function would provide some bias in the cost comparison. Therefore, I included two cost estimates for the *divide-and-conquer* algorithm: one estimate

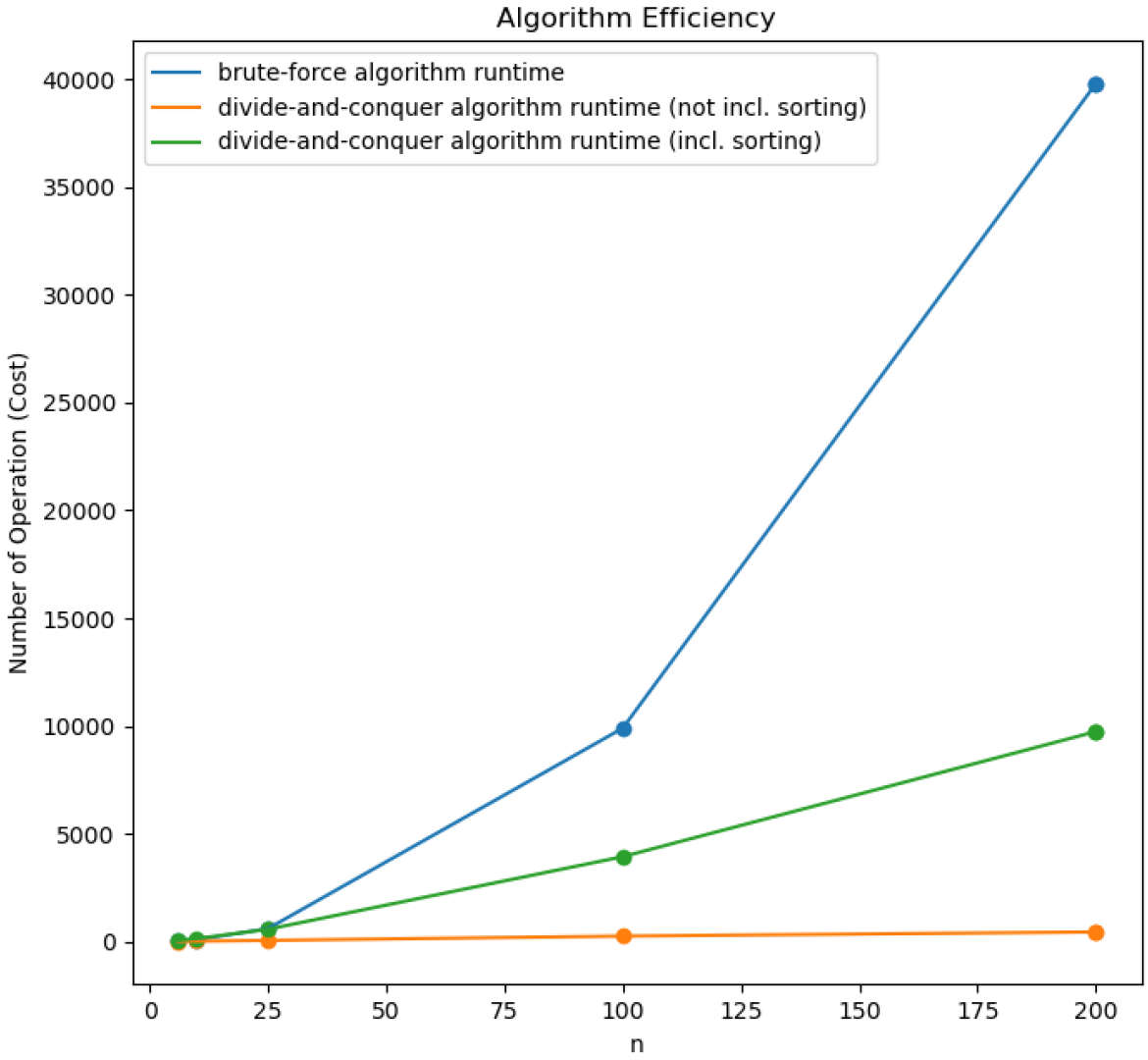


*Figure 1 - Runtime costs for one series of trace runs evaluated up to n = 200.*

(column C) gives the cost for the algorithm that only includes the number of calls to the distance function and the other estimate (column D) gives the cost for the algorithm including both the number of calls to the distance function and the runtime complexity of merge-sort twice (merge-sort has a runtime of *T(n lg n)* for average, worst, and best cases).

If we only consider the number of calls made to the *get_distance_between_pairs( )* function, we immediately see from Column C that the *divide-and-conquer* algorithm garners a significantly smaller runtime in comparison to its *brute-force* algorithm counterpart in Column B. Without knowing anything about the algorithm itself, we can automatically assume that the *divide-and-conquer* strategy performs some additional management and computation within its subroutines in order to facilitate less calls to the distance function. This extra management comes at a price, though. If we consider the number of operations it takes to sort the array each time inside the *divide-and-conquer* algorithm, we see that the extra housekeeping it performs to eliminate the

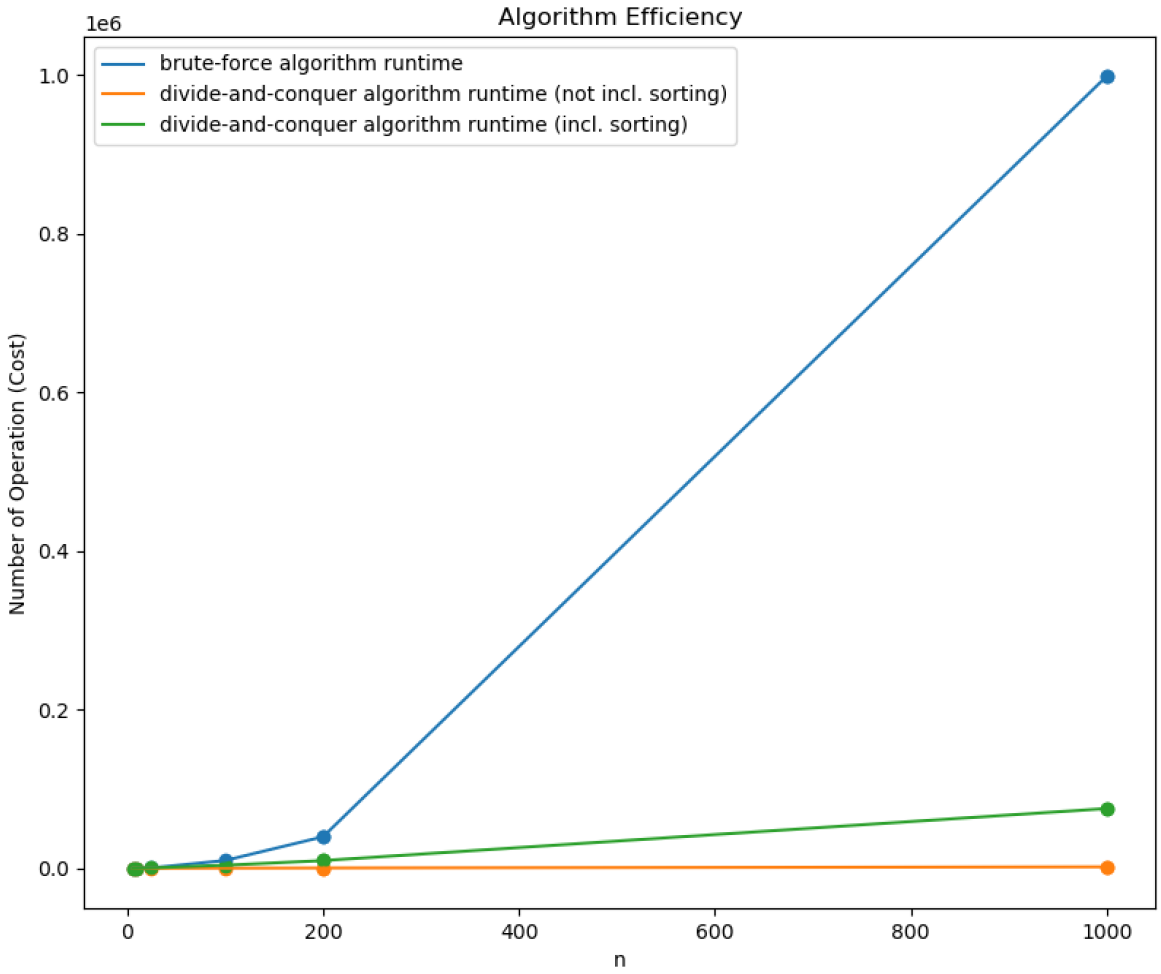


*Figure 2 - Runtime costs for one series of trace runs evaluated up to n = 1000.*

calls to the distance function are taxed by the increased operational count to sort the input. So, although less calls are made to the primary mathematical operation ( *get_distance_between_pairs( )* ), we make up for it in added runtime with the sort. Column D addresses this notion by showing the runtime of the *divide-and-conquer* algorithm with the operational count of the merge-sort methods included. Now, if we consider the values in Column D with their *brute-force* counterparts in Column B, we see that we actually achieve a *worse* runtime than the *brute-force* method for low values of *n.* As *n* scales, we begin to notice the true value of presorting the data since the *divide-and-conquer* method out-performs the *brute force* method after *n = 6.*

This principle is easy to see visually. Consider Figure 1 in which we have a series of trace runs for both algorithms up to *n = 200.* In this figure, we confirm that the *divide-and-conquer* method actually performs worse than the *brute-force* method for *n = 6,* but it is then sharply crossed by the *brute-force* line for successive values of *n,* and the rate at which the cost of the *brute-force* algorithm diverges is significant. If we consider Figure 2, we observe a similar trend in a different trace run as well. We also see how much more extreme the divergence of the *brute-force* cost is for a much higher value of *n* in comparison to both cost metrics of the *divide-and-conquer* algorithm.

In order to effectively evaluate the cost of both the *brute-force* and *divide-and-conquer* functions in an unbiased manner, it is important to consider the cost of the *divide-and-conquer* algorithm both with and without the sorting methods. The above figures show that the *divide-and-conquer* algorithm significantly outperforms the *brute-force* method in most cases regardless of whether or not sorting is considered, but we do see that the costs to sort add a significant amount of runtime to an otherwise more efficient algorithm. Mathematically, we validated this in the preceding sections showing the contrast in runtime costs for both algorithms.

## Lessons Learned

*Lab 0* helped me better appreciate discipline behind good algorithm design. The biggest lesson I learned is to adequately consider the maintenance behind an algorithm that returns a generally better runtime but necessitates sorting in order to function. At first thought, I assumed that the operational cost to sort a dataset multiple times in an algorithm would contribute to a significantly higher runtime cost to the algorithm than what manifested in the *divide-and-conquer* algorithm. Until researching more through the textbook[1], I originally only considered approaches for a more efficient algorithm that did not require sorted datasets. Scaling the trace runs helped me clearly see that, at least in this scenario, sorting the data contributed much less weight than facilitating a brute-force method.

Another point that really resonated with me throughout the lab was just how powerful the *divide-and-conquer* strategy is. By recursively downsizing the problem into smaller subproblems, and

then using the solutions to those subproblems to scale up to a final solution, we follow a similar approach as merge-sort and binary search – two very powerful and efficient algorithms. This strategy is extremely insightful in algorithm design and something I see continues to hold value as we study reduction in P/NP-completeness. I originally approached the problem with a divide-and-conquer strategy, but the lab only further strengthened my enthusiasm for reductionist techniques by pointing out where they fail (at small scales) and where they truly do optimize computational resources.

## Enhancements

The program provides 5 major enhancements (along with many other minor enhancements) on top of the original requirements.

1) The first major enhancement involves the automatic removal of duplicate points and covering of corner cases. When returning the *m* closest points among *n* total points, the algorithm will naturally consider the distance between A-B and B-A as two separate distances and present them both when sorting. I optimized my sorting methods to account for this and only include one distance for the pair, removing the second through a duplicates filter. Additionally, the *brute-force* and *divide-and-conquer* methods naturally want to find the distance between one point and itself due to the dual for-loop in the *brute-force* algorithm and the recursion calls in the *divide-and-conquer* algorithm. Trivially, this will always be zero and will always be returned as the shortest distance. As such, I optimized the algorithms to filter out these corner cases when finding the closest pair of coordinates withing a series of coordinates.

2) The second major enhancement incorporated time delays. When the program is conventionally executed, the command prompt flashes a plethora of data and finishes executing in under a second. The user is then forced to go back and read through the steps executed. *FranceLab0* incorporates time delays to allow the user to read information in the command prompt as processing steps are completed. For example, when the program starts up, a time delay of 3 seconds allows the user to know whether or not the input files were detected. Additionally, when the program successfully imports the I/O files, this information is logged to the command prompt and a time delay of 3 seconds halts processing so that the user can take in this information accordingly. After the data for each trace run is configured, processing is briefly paused to allow the user to visualize the operators, operands, and result. Finally, after each cost and correctness run finishes evaluation, a time delay of 1.5 seconds briefly holds processing for each evaluated set of points as well to allow the user to see whether or not that step failed.

3) The next major enhancement revolves around error handling. If an input file contains illegal characters (non-numeric numbers and non-decimals), the program notifies the user through both the console and the output file. However, instead of simply letting the user

know that the expression contained invalid characters, they are notified of exactly which characters were illegal in their expression. Processing then continues with the points that were successfully read in, if any.

4) The fourth major enhancement builds a summary metric for the user through a graph. At the end of the program, a summary of the runtime costs evaluated for each trace run are consolidated and displayed in a graph. This makes it exceptionally simple for the user to visualize how well the two algorithms performed in comparison to each other.

5) The final major enhancement involves aesthetics. It is one thing to satisfy the project requirements by dumping all of the information to the console and/or output file, but I felt that it was important to make things easier to follow and interpret as the program executed. I tried to discretely separate cost runs form correctness runs and intentionally segregate trace run data for different levels of *n.* I also tried to communicate data more effectively by displaying output like *"distance 0.5 from point (a, b) to point (c, d)"* instead of the simpler *"0.5, (a, b) to (c, d)".* These principles were followed throughout both the console output and output file, the console output being a duplicate of what was written to the file.

6) A sixth minor enhancement involves an approach with a genetic algorithm. The file *genetic.py* contains logic for a genetic algorithm that finds the closest point among a series of points. While the file is not used in the lab, a genetic algorithm was my first (and failed) approach of developing an "efficient algorithm" as specified in the lab handout. I included it due to this and because I briefly mention it in the analysis document. If desired, the genetic algorithm may be run with the command *python Lab0/genetic.py.*

## Future Optimizations

The sorting algorithm has some room for optimization within the *divide-and-conquer* function. Right now, the algorithm is *merge-sort* which runs in $T(n) = n \lg n$ time in all cases, but there is room for improvement with another algorithm such as *timsort.*

## Conclusion

The *FranceLab0* is a Python module performs mathematical operations on points within a 2D Cartesian coordinate space in order to find the closest pair of points in a series of points. The program evaluates the algorithmic efficiency for two search algorithms – a *brute-force* algorithm and a *divide-and-conquer* algorithm; the program automatically generates trace runs and performs cost analysis for both. The closest *m* pairs of points within a series of *n* points are

returned to the user for values of 3 and 5 for *m*. Results from the program are output for different sizes of datasets through correctness runs and cost runs. Finally, a summary of the results is computed and displayed through a graph that shows the asymptotic runtime costs for both algorithms under different dataset sizes of *n*.

# References

The following items were used as references for the construction of this project.

1) Cormen, T. H., & Leiserson, C. E. (2009). Introduction to Algorithms, 3rd edition.

2) Miller, B. N., & Ranum, D. L. (2014). Problem solving with algorithms and data structures using Python (2nd ed.). Decorah, IA: Brad Miller, David Ranum

3) Merge Sort. GeeksforGeeks. (2021, June 20). https://www.geeksforgeeks.org/merge-sort/.

4) QuickSort. GeeksforGeeks. (2021, June 20). https://www.geeksforgeeks.org/quick-sort/.