





# IFT2015 ASSIGNMENT 1

June 12, 2023

Massimo Pietracupa (20206087) and Vincent Hoang (20183549)

## 1: Auto-evaluation: Specify if your program works correctly, partially or does not work at all.

Our program seems to be working correctly. When we compare the outputs that are received to those that were provided by the lab instructor, the outputs matched exactly:

Output of our Program	Sample output provided
 output0.txt - Notepad	 resultat_camion_entrepot0.txt - Notepad
File Edit Format View Help	File Edit Format View Help
Truck Position: (45.5532,-73.5968)	Truck position: (45.5532,-73.5968)
Distance:0.0 Number of boxes:0 Position:(45.5532,-73.5968)	Distance:0 Number of boxes:0 Position:(45.5532,-73.5968)
Distance:678.5 Number of boxes:0 Position:(45.5593,-73.5966)	Distance:678.5 Number of boxes:0 Position:(45.5593,-73.5966)
Distance:839.6 Number of boxes:0 Position:(45.5569,-73.6062)	Distance:839.6 Number of boxes:0 Position:(45.5569,-73.6062)
Distance:1321.0 Number of boxes:0 Position:(45.5487,-73.5811)	Distance:1321.0 Number of boxes:0 Position:(45.5487,-73.5811)
Distance:1321.3 Number of boxes:0 Position:(45.565,-73.5988)	Distance:1321.3 Number of boxes:0 Position:(45.565,-73.5988)
Distance:2078.6 Number of boxes:0 Position:(45.5352,-73.5896)	Distance:2078.6 Number of boxes:0 Position:(45.5352,-73.5896)
Distance:2081.5 Number of boxes:0 Position:(45.5686,-73.612)	Distance:2081.5 Number of boxes:0 Position:(45.5686,-73.612)
Distance:2221.0 Number of boxes:42 Position:(45.5412,-73.6196)	Distance:2221.0 Number of boxes:42 Position:(45.5412,-73.6196)

The same is true for all other example files.

## 2: Theoretical temporal complexity analysis (worst case, big O notation). Analysis has to be done on the solution pseudo code or on the developed Java code

Our program can be separated into five main components as follows:

1. Read in file while taking into account the largest storage location.
2. Perform initial swap to place largest storage location at beginning of list.
3. Single Loop through storage location list to calculate, and store, the distances of each storage location.
4. Perform a quicksort on the list with respect to distances.
5. Single Loop through storage location list to output data into output file (while performing boxes calculation).

Let us take a look at the theoretical temporal complexity analysis of each section:

### Section 1:

Reading in the file and processing each storage location to be stored, will have a fixed time per storage location. With number of storage locations being represented as  $n$ , we can say that the time complexity for this section will at worst case be  $O(n)$ .

```

61 public static CargoInfo processFile(String filePath)
62 {
63     CargoInfo cargoobj = new CargoInfo();
64     try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
65
66         String line;
67         Pattern pattern = Pattern.compile("\\((-?\\d+\\.\\d+),(-?\\d+\\.\\d+)\\)");
68         int largestBoxStorage = 0;
69         List<StorageInfo> storageLocationsObj = new ArrayList<>();
70         int counter = 0;
71         while ((line = reader.readLine()) != null) {
72             if (counter == 0)
73             {
74                 String[] parts = line.split(" ");
75                 cargoobj_boxesToTransport = Integer.parseInt(parts[0]);
76                 cargoobj_remainingBoxes = Integer.parseInt(parts[0]);
77                 cargoobj_truckCapacity = Integer.parseInt(parts[1]);
78             }
79             else
80             {
81                 String[] parts = line.split(" ");
82                 for (int i = 0; i < parts.length; i += 2)
83                 {
84                     StorageInfo storageobj = new StorageInfo();
85                     storageobj_boxStorage = Integer.parseInt(parts[i]);
86                     if (storageobj_boxStorage > largestBoxStorage)
87                     { ...
88                     }
89                     Matcher matcher = pattern.matcher(parts[i+1]);
90                     if (matcher.find())
91                     { ...
92                     }
93                     // Store into destination list
94                     storageLocationsObj.add(storageobj);
95                 }
96             }
97             counter += 1;
98         }
99
100         cargoobj.storageLocations = storageLocationsObj;
101     } catch (IOException e) {
102         e.printStackTrace();
103     }
104 }

```

## Section 2:

In the second section, we simply perform 1 swap, to place the largest storage location at the beginning of the list. Regardless of the size of the storage locations, this will always be one operation, hence we can say that at worst case the time complexity will be  $O(1)$ .

```

public static void Algorithm(CargoInfo cargoobj)
{
    // Replace the largest storage location at top of list
    try {
        FileWriter fileWriter = new FileWriter(cargoobj.outputFile);
        BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

        // Start by swapping the largest index to slot 1 - which was found upon reading file
        2. qsswap(cargoobj.storageLocations, 0, cargoobj.indexLargestStorage);
        cargoobj.storageLocations.get(0).distance = 0;

        String text = "Truck Position: (" + cargoobj.storageLocations.get(0).latitude + "," + cargoobj.storageLocations.get(0).longitude + ")\n";
        System.out.println(text);
        bufferedWriter.write(text);

        // Loop through the storage locations to calculate the distance
        for (int j = 1; j < cargoobj.storageLocations.size(); j++)
        {
            3. cargoobj.storageLocations.get(j).distance = calculateDistance(cargoobj.storageLocations.get(0).latitude,
                cargoobj.storageLocations.get(0).longitude,
                cargoobj.storageLocations.get(j).latitude,
                cargoobj.storageLocations.get(j).longitude);
        }

        // Perform the quick sort algorithm on the storage locations
        4. quickSort(cargoobj.storageLocations, 0, cargoobj.storageLocations.size() - 1);

        // Print out the Data
        for (int i = 0; i < cargoobj.storageLocations.size(); i++)
        {
            cargoobj.remainingBoxes -= cargoobj.storageLocations.get(i).boxStorage;
            text = printData(cargoobj.storageLocations.get(i).distance, calculateRemainingStorage(cargoobj),
                cargoobj.storageLocations.get(i).latitude, cargoobj.storageLocations.get(i).longitude);
            bufferedWriter.write(text);
            5. if (calculateRemainingStorage(cargoobj) != 0)
            {
                // We've reached the end of storage locations and still have remaining boxes - exit
                bufferedWriter.close();
                return;
            }
        }
    }
    catch (IOException e) {
        System.out.println("An error occurred while writing to the file: " + e.getMessage());
    }
}

```

### Section 3:

In this section we perform a single pass over the storage location list and perform distance calculations with the first entry. This section should increase based on the number of storage locations resulting in a time complexity of  $O(n)$ ,

### Section 4:

We perform a quicksort on the list in order to order the entries in increasing order based on their distances. The quicksort algorithm performs partitioning and rearranges the array around a pivot. With the pivot in the correct sorted position (with all elements before it being smaller – and all elements after it being greater), we can repeat this recursively until the entire array becomes sorted. This is a divide and conquer technique, which has an average time complexity of  $O(n \log n)$ , with a worst case scenario at  $O(n^2)$ , when the pivot selection is unbalanced.

### Section 5:

In the final section, we simply loop over the array a final time to return the results into the output file. In a scenario where the number of storage locations does not completely deplete the storage of the truck, this will loop  $n$  times resulting in a time complexity of  $O(n)$ .

If we combine the time complexities of each section we can deduce that the time complexity for our entire program can be computed as followed:

*Average Case:*

$$1 + n + n \log n + n = 1 + 2n + n \log n \Rightarrow n \log n$$

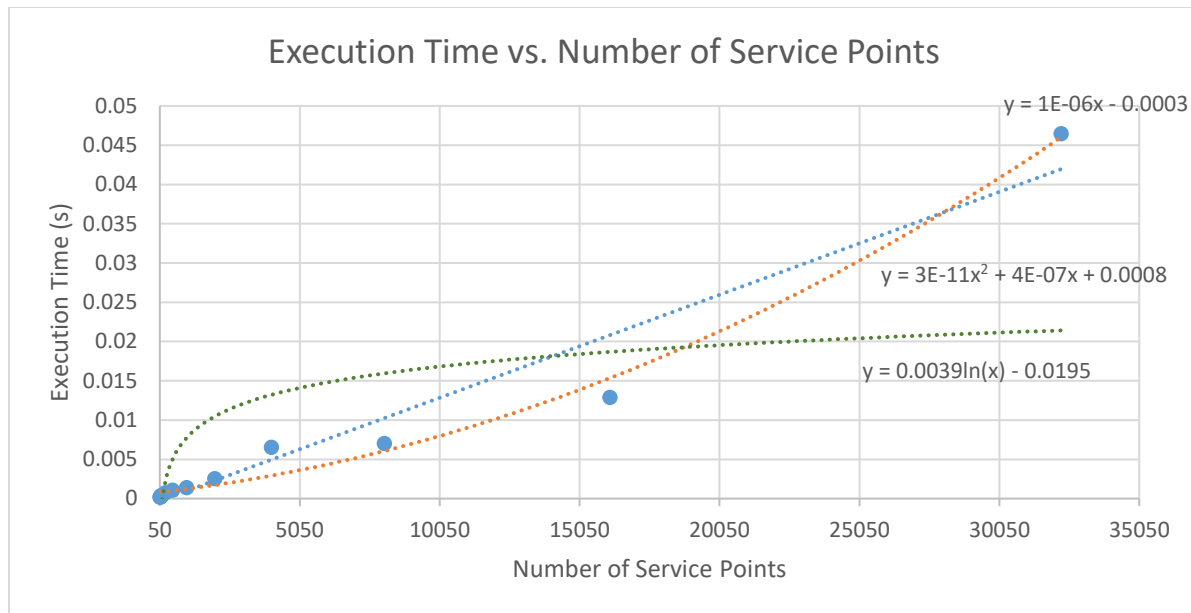
*Worst Case:*

$$1 + n + n^2 + n = 1 + 2n + n^2 \Rightarrow n^2$$

Hence we can deduce that our program will perform somewhere in between  $n \log n$  and  $n^2$  during normal execution.

### **3. Empirical temporal complexity analysis. Graphic demonstrating the algorithm running times for different sizes of the input.**

# of Service Points	Running Time (seconds)
62	0.0001763
96	0.000383
68	0.000205
64	0.0001957
90	0.0002939
86	0.0002649
254	0.0007473
506	0.0010805
1010	0.001401
2018	0.0025421
4034	0.00651
8068	0.0070026
16130	0.0128811
32258	0.0464765



We have created a dataset with an increasing amount of service points and have plotted their running times in a graph. We have included a linear trendline ( $O(n)$ ) as a reference line for our chart. It is clear that the execution time trend is not linearly dependent on the input size ( $n$ ). We can also see that it's growth is not drastic enough to consider a quadratic growth ( $n^2$ ) though it seems a little steeper than  $n\log n$ . As a result, the theoretical assessment seems to be appropriate and we can confirm that it does seem to fall somewhere between  $n\log n$  and  $n^2$ .