DEPARTAMENTO DE ENGENHARIA
INFORMÁTICA

MDISC Report

June 18, 2022

1DI, Group 40

1191296, Gabriel Gonçalves

1191369, Tiago Leite

1211314, João Durães

1211304, Francisco Bogalho

1210805, António Bernardo

isep Instituto Superior de Engenharia do Porto

Sandra Luna (SLU)

Index

1. Sorting clients by arrival time, or by leaving time

Introduction

In the sprint D of the Integrative project, the MDISC course entered in two User Stories (US) to be developed by our team. The US 16 wanted us to develop a brute-force algorithm to find the maximum subsequence in a given array.

The US 17 required to our team to develop two sorting algorithms. Our team chose the quick sort and the bubble sort algorithms. The first one is theoretically the faster and is recursive algorithm, the second one is theoretically slower.

We will analyze bought of them with runtime tests for different input sizes, and then the worst-case time complexity.

What is Bubble Sort?

Bubble Sort Algorithm is one of the simpler but isn't very efficient. It sorts elements in ascending order by simply comparing adjacent values two to two and placing them in ascending order (in case they aren't in that order).

What is Quick Sort?

Quick Sort is a sorting algorithm belonging to the divide-and-conquer group of algorithms. This type of algorithms recursively breaks down a problem into two or more subproblems of same type, making them simpler to solve.

This algorithm has the best results when working with large arrays of data.

Runtime tests for inputs of varying sizes

In this section, we will test the two algorithms with runtime tests for inputs of varying sizes.

Bubble Sort

For 7049 inputs	
Test number	**Time elapsed (seconds)**
1	375.56
2	370.18
3	372.06

Average: 372.6s

For 3524 inputs	
Test number	**Time elapsed (seconds)**
1	94.14
2	96.03
3	93.46

Average: 94.543s

For 1762 inputs	
Test number	**Time elapsed (seconds)**
1	23.44
2	23.35

3	23.92

Average: 23.57s

For 881 inputs	
Test number	**Time elapsed (seconds)**
1	5.92
2	5.86
3	5.91

Average: 5.897s

For 440 inputs	
Test number	**Time elapsed (seconds)**
1	1.47
2	1.47
3	1.49

Average: 1.477s

For 220 inputs	
Test number	**Time elapsed (seconds)**
1	0.39
2	0.39
3	0.38

Average: 0.387s

Quick Sort

For 7049 inputs	
Test number	Time elapsed (seconds)
1	3.84
2	3.75
3	3.84

Average: 3.81s

For 3524 inputs	
Test number	Time elapsed (seconds)
1	1.16
2	1.18
3	1.15

Average: 1.163s

For 1762 inputs	
Test number	Time elapsed (seconds)
1	0.44
2	0.45
3	0.47

Average: 0.453s

For 881 inputs	
Test number	Time elapsed (seconds)

1	0.19
2	0.17
3	0.20

Average: 0.187s

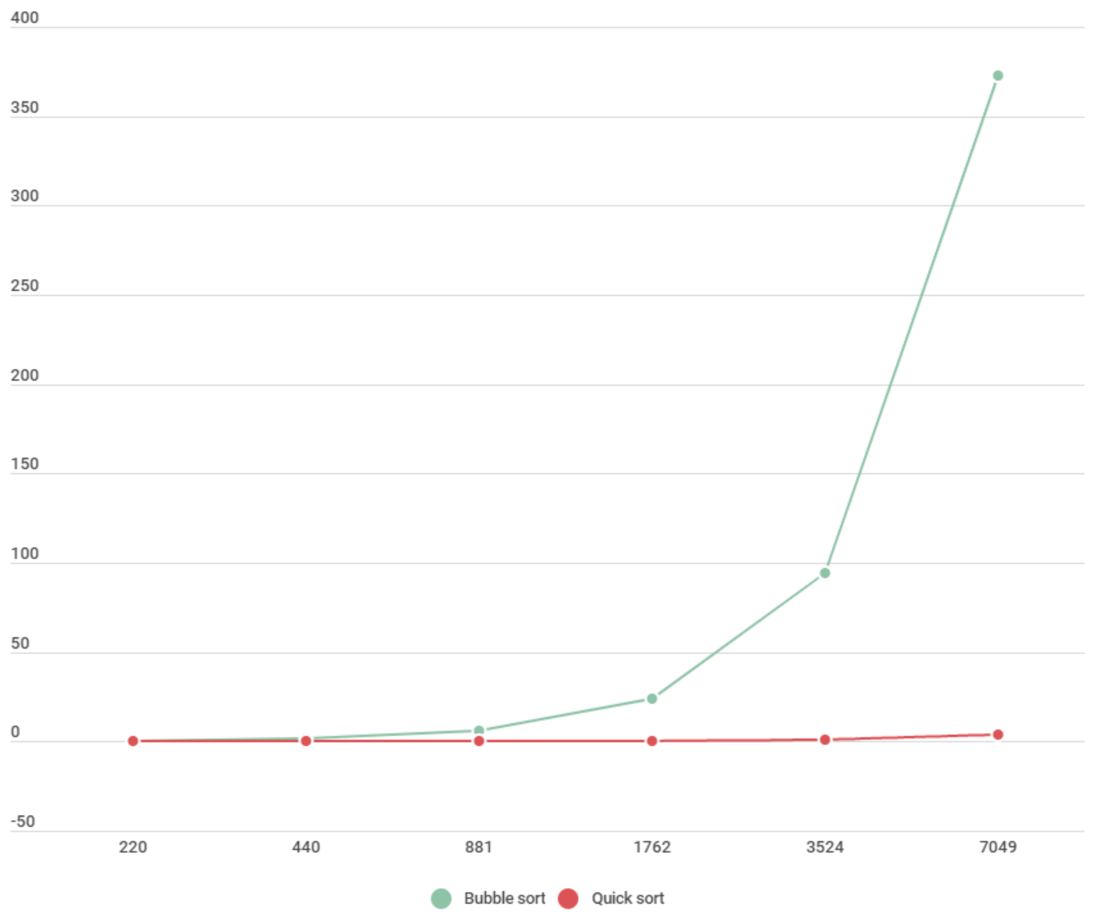
For 440 inputs	
Test number	Time elapsed (seconds)
1	0.12
2	0.07
3	0.11

Average: 0.10s

For 220 inputs	
Test number	Time elapsed (seconds)
1	0.07
2	0.04
3	0.08

Average: 0.063s

With this varied-length time intervals, we can build the following graphical representation:



We can see that the Quick Sort algorithm is considerably faster than the Bubble Sort one. The time elapsed in both algorithms increase when the input length increase, but we can observe that the Bubble Sort increases exponentially when the input grow. Otherwise, the Quick Sort, increase when the input length increase, but it’s much less than the other.

From this test, we can see that the Quick Sort is a better algorithm to sort data, specially when the input length tends to be high.

Worst-case time complexity analysis

Bubble Sort

```
Line                           Code
 1    procedure bubbleSort(a[1], a[2], …, a[n]:real)
                   for i:= 1 to n-1
                       for j:= 1 to n - i
                           if a[j] > a[j+1] then
                              swap a[j] and a[j+1]
```

Analyzing line by line the worst time complexity of the Bubble Sort algorithm, we can see the following.

Line	Number of operations made
1	(Method header)
2	n-1
3	$(n-1)+(n-2)+(n-3)+\cdots+\big(n-(n-1)\big)=(n-1)+(n-2)+\cdots+1=\frac{n-1}{2}*n$
4	$\frac{n-1}{2}*n$
5	Not a primitive operation

So, making this analyze, we can conclude that the worst-time complexity is $O(n)+O(n^2)+O(n^2)=O(n^2)$. This algorithm has polynomial complexity.

Quick sort

```
Line                           Code
 1    procedure quickSort(a[1], a[2], …, a[n]:real, low:int, high:int)
 2                 if low < high then
 3                     int p = partition(a[1], a[2], …, a[n], low, high)
 4                     quickSort(a[1], a[2], …, a[n], low, p-1)
 5                     quickSort(a[1], a[2], …, a[n], p+1, high)
```

```
Line                           Code
 1    procedure partition(a[1], a[2], …, a[n]:real, low:int, high:int)
 2                 int p = low, j
 3                 for j := low + 1 to high
 4                     if arr[j] < arr[low] then
 5                         swap a[p+1] to a[j]
                   swap a[low] to a[p]
```

	Number of operations
1 | 2 | 3 | 4 | 5 | … | n	n
2 | 3 | 4 | 5 | … | n	n – 1
3 | 4 | 5 | … | n	n – 2
4 | 5 | … | n	n – 3
5 | … | n	n – 4
… | n	…
n – 1 | n	1

Quick Sort can have, theoretically, the worst-case of $O(n^2)$.

2. Evaluation of the effectiveness of the vaccination center's response

Introduction

In order to analyze the performance of the vaccination center, the center coordinator intends to check the number of clients arriving/departing the center at multiple time intervals previously chosen in a given day, in order to verify the effectiveness in the respective time intervals.

For this, a brute force algorithm was implemented that examines all contiguous sublists and determines the one that has the maximum sum, which is the interval where the center was least responsive.

Below we can visualize the algorithm mentioned in pseudocode:

```
Procedure findMaxSumSublist(integer, listToBeAnalyzed[1], listToBeAnalyzed[2],..., listToBeAnalyzed[n])
            startIndex := 0
            endIndex := 0
            maximumSum := 0
            arrayLength = n
            for i:= 0 to arrayLength
                currentSum := 0
                for j:= i to arrayLength
                    currentSum = currentSum + listToBeAnalyzed[j]
                    if currentSum > maximumSum
                       maximumSum := currentSum
                       startIndex := i
                       endIndex := j
            return array[] = listToBeAnalyzed [startIndex],..., list[endIndex]
```

Runtime tests for inputs of varying sizes

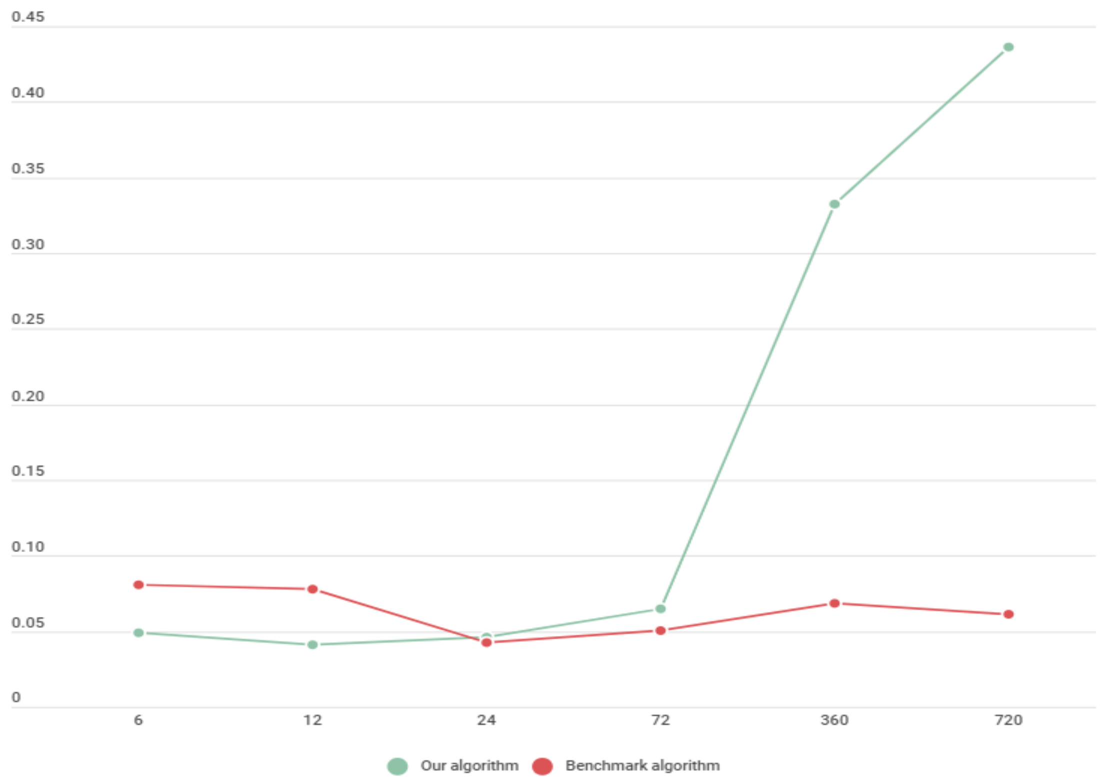
In order to evaluate the execution time of our algorithm, a comparison was made with a reference algorithm (Benchmark Algorithm) for which variable sizes were used.

The variable sizes used were 6, 12, 24, 72, 360, 720.

For this analysis, several executions of the 2 algorithms were carried out and the execution times obtained were recorded, which are shown in the following table:

Size	Our algorithm	Benchmark algorithm
6	0,05	0,08
12	0,04	0,08
24	0,05	0,05+
72	0,06	0,05
360	0,34	0,06
720	0,44	0,06

The following graph shows a comparison between the two algorithms. A quick analysis shows that our algorithm is more effective for small sizes, but less so for large sizes. The Benchmark Algorithm, on the other hand, is exactly the opposite.



Worst-case time complexity analysis

The worts case time complexity analysis for the findMaxSumSubList procedure is as follows:

The primitive operation executed in the code are:

- **A** (Assignment) associate a value to a variable;
- **I** (Increment) increment a value or a variable to another variable;
- **C** (Comparison) a boolean operation is done to obtain a logical value;
- **OP** (Operation) an arithmetic operation;
- **R** (Return) return the procedure result

The table presented below is a summary of the complexity analysis.

Line	Operations
1	1A
2	1A
3	1A
4	1A
5	(n+1)A + (n+1)I + (n+1)C
6	nA
7	n((n+1) A + (n+1)I + (n+1)C)
8	(n*n)OP+(n*n)A
9	n*C
10	nA
11	nA
12	nA
13	1R

In the first two lines of code, the variables startIndex and endIndex, respectively, are initialized to o. In the third line of code we initialize the variable maximumSum to zero.

In the first two lines of code, the variables startIndex and endIndex that indicate the lower index of the input list and the upper index of the list, respectively, are initialized to 0. Next, the value 0 is assigned to maximumSum.

Then we insert a "for" loop on line 5. We initialize the i to zero, which will be incremented up to the size of the array. In this loop I start by assigning zero to the currentSum value on line 6 and then another "for loop that initializes the j that is assigned the value of i and which will be incremented up to the size of the array in line 7.

In line 8 there is an assignment of the currentSum variable to its own value with the addition of the value of the array to be analyzed at position j.

Then there is a comparison between the current sum and the maximum sum. If the current sum is greater than the maximum sum, 3 assignments are performed.

First, the maximum variable sum is assigned to the current sum and the start index and end index are assigned to the variable i and j respectively.

Finally, in line 13, an array is returned with the contiguous sublist of the maximum sum found with the execution of the algorithm.