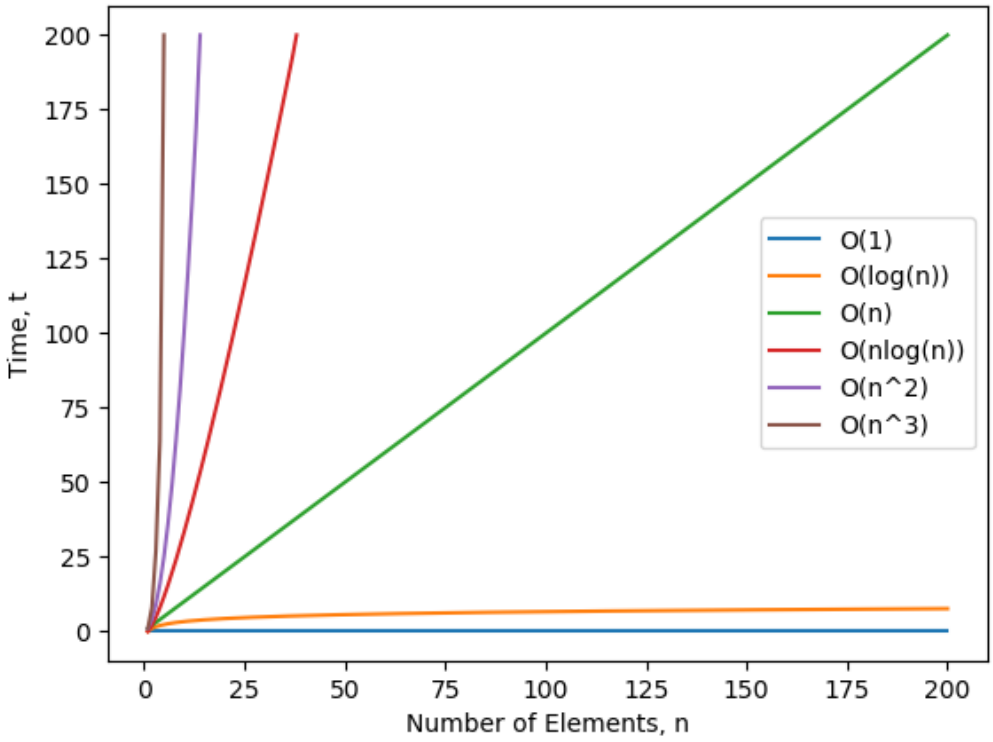
Algorithm Analysis



Diogo Teixeira – 1200904
Miguel Ferreira – 1211488
Pedro Campos – 1211511
Lucas Gonçalves – 1211601
Nuno Cunha – 1211689

Index

Índice

Introduction

In this document we will comprehend and explore in a more depth way the algorithms used in the process of developing certain user stories from the integrative project (US16, US17) and its complexity, more specifically their time complexity.
As said before an algorithm will be implemented for US16. To be more specific, a brute-force algorithm, with the purpose of evaluating and analysing the performance of a given vaccination center on how they respond to the arrival of multiple new SNS users during the multiple intervals of times of a certain day. This brute-force algorithm will be compared to a benchmark algorithm with variable size of inputs (24,36,72,144,720), in order to see its asymptotic behaviour.
In the other hand for US17 we will not implement any brute-force algorithm since we want to sort some data that as be given to us from a legacy system and for this usage a brute-force is clearly not the best option. Instead, we will implement two different sorting algorithms and let the application user choose which one they desire to use to sort the given information. Either sorting by arrival time or by the center leaving time. The documentation process of this last two mentioned algorithms will be similar to the brute-force, we will show the pseudo-code and proceed to analyse the time complexity, but instead of comparing the algorithms with a benchmark algorithm, we will show the graphics that represent the running time of each algorithm for both types of sort, with variable size of the input number.

Brute-force Algorithm (US16)

As said in the introduction, this is the core algorithm implemented in US16 and its usage is to evaluate and analyse the performance of a given vaccination center on how they respond to the arrival of multiple new SNS users during the multiple intervals of times of a certain day.

Brute-force algorithm:

```
procedure bruteForce (a[n] : integer)
        m := -2147483648
        s := 0
        f := 0
        for i := 1 to n
                sum := 0
                for j := 1 to n
                        sum := sum + a[ j ]
                        if sum > m
                                s := i
                                f := j
return m
```

Explanation:

This brute force is going to work for an array of integers and will calculate the maximum sum of a contiguous subarray from this original one.
We will have a variable represented as *m* that will be the variable that will be updated with the value of the maximum sum during this procedure. This variable is instantiated with the value of the minimal integer possible (-2147483648).
Like we made in the first line, the variables in the second and third line will store during the procedure the values of the index of the start and end of the subarray, respectively.
To brute-force the process of getting the maximum sum in a contiguous subarray from this initial one, we will have two *for* cycles, that will be used to continuously add the values of the array while it is traversed.
This operation of adding the values will result in the *sum* variable being compared with the *m* variable to verify if it is bigger and if so, its

value is assigned its value to the *m* variable and the start and end index values assigned to *s* and *f* variables, respectively.
With this said, we can confirm that this brute-force will work and will stop, because we will traverse all the possible combinations of subarrays sums and return the biggest one. To reinforce the certainty that the algorithm works, we tried the inputs given in the MDISC page and all the outputs were the desired.

Brief note:

To solve the user story, we used the brute-force algorithm presented above but with proper classes to fulfil all the integrative project requirements, having this no implication in the time complexity so the analysis is equal to this algorithm with both types of classes.
Being this said, we decided to analyse this brute-force algorithm with the usage of different values from the csv file given to us, making the analysis process much easier to test the input values. And then compared to the running times of the benchmark algorithm (Kadane's algorithm) for the same number of inputs, but this time we used integers since the benchmark algorithm was made for integer inputs.

Time complexity:
For our time complexity we will use the notation used in the MDISC notebook. (A) attribution, (I) increment, (C) comparation, (Op) Operation and (R) return.

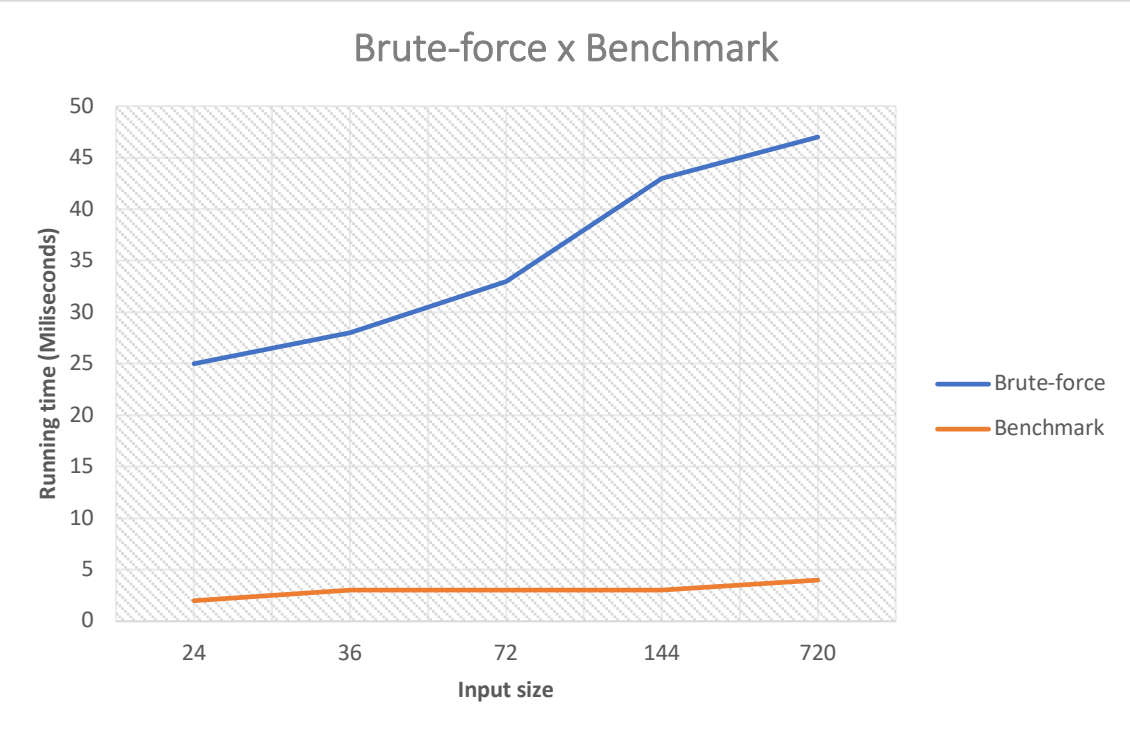
Lines	Brute-force	Big-O
1st	1A	O (1)
2nd	1A	O (1)
3rd	1A	O (1)
4th	(n)A or I + (n) C	O (n)
5th	(n)A	O (n)
6th	(n)A or I + (n) C	O (n)
7th	(n)A or Op + (n) A	O (n)
8th	(n) C	O (n)
9th	(n)A	O (n)
10th	(n)A	O (n)
11th	1R	O (1)
Total	21n^2 + 4	O (n^2)

In order to obtain the time complexity of this brute-force algorithm, we will use some of the theorems present in the MDISC notebook together with the values of the table above.
We already explained the logic behind the algorithm and represented the cost and complexity of each line in the table above.
The attributions outside both *for* loops will sum all together and we will get the value 4 presented in the total in the table above.
In the *for* loops, we must be cautious since we have a nested for loop, so the complexity of the two *for* loops will be obtain by multiplying the complexity of the "outside" *for* loop by the nested one. Doing this we obtained the ((2n + 1) (5n + 2)) value in the total column of the table above.
Now talking about Big-O notation, this brute-force algorithm will have O(n^2) as time complexity, this is justified with the use of two theorems:
"$f(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$, with $a_i \in \mathbb{R}$, for all the i values. So, $f(x)$ is O (x^n)".
"Being f_1 and f_2 functions that $f_1(x)$ is O$(g_1(x))$ and $f_2(x)$ is O $(g_2(x))$. So $f_1(x)f_2(x)$ is O $(g_1(x)\ g_2(x))$".
The first theorem justifies why we will get O (n) when summing all the complexity values inside the nested for loop, why we get O (1) when we sum all the attribution values and the most important, why we get O (n^2) as the total time complexity.
The second one justifies the fact that we obtain O (n^2) when multiplying both complexities of the *for* loops, O (n^2).
In a small note, this is a brute-force algorithm like said before, so this time complexity analysis will always be the worst-case one, because we don't have any middle code breakout points. With this being said, we can also add that the total number of iterations of the *for* loops can be represented by the expression below. This is given by the fact that the "outside" *for* loop will run *n* times the second one, that is also running *n* times.

$$\sum_{i=0}^{n}(n)$$

Running time test (different inputs):

In this part we will test our brute-force algorithm and the given benchmark algorithm (Kadane's algorithm), for different sizes of inputs, 24, 36, 72, 144 and 72.



Brute-force conclusion:

To conclude, we can see by the graphic above that the complexity of the benchmark algorithm (Kadane's algorithm) O(n) is notably better than the brute-force with O(n^2) when the input size increases, and this only will be more and more notable while the input size increases.

Sorting algorithms (US17)

We introduced this topic in the introduction, this user story concerns the fact that we will have to import some data from a legacy system.
To do this we will implement a functionality that will save all the given data to the current system, but for the MDISC component we will focus on the fact that this data will be sorted either by arrival time or leaving time.
We have done this with the implementation of two different sorting algorithms that will work both for the two possible types of sort.

Brief note:

Like we said back in the brute-force, we used proper classes to fulfil the requirements of the integrative project, so we adapted our two sort algorithms originally made for integers to the class *csvInfo* that is a class that will contain all the information given to us by the csv file. With this we want to reinforce the idea that for both types of classes the time complexity will be the same because all the operations made have the same complexity.
For the input test we will also use the data given to us in the csv file, but these two chosen sorting algorithms will be tested for bigger input sizes.

Bubble sort algorithm:

```
procedure bubbleSort ( a[n] : CsvInfo)
        for i := 1 to n-1
                for j := 1 to n-1
                        if a[ j + 1] < a[ j ]
                                swap a [ j ] and a[ j + 1]
return a
```

Explanation:

The first algorithm that we have chosen is the bubble sort algorithm. This algorithm will work with two *for* loops like our brute force, so the analysis will have some similarities.
It will iterate over all the members of the array and will successively verify if the next member of the array is smaller than the current one, if so the algorithm will swap their positions.
We need the two for cycles in order to iterate all members n^2 times, because in the worst-case all the members are in the wrong order, and we will need to change all the members n-1 times. And we need the if clause to compare both values and see if it is indeed needed the change of postions between both.

Time complexity:

For our time complexity we will use the notation used in the MDISC notebook. (A) attribution, (I) increment, (C) comparation, (Op) Operation and (R) return.

Lines	Bubble sort	Big-O
1st	(n-1)A or I + (n-1)C	O(n)
2nd	(n-1)A or I + (n-1) C	O(n)
3rd	(n-1)A + (n-1)A + (n-1)C	O(n)
4th	3((n-1)A + (n-1)Op)	O(n)
5th	1R	O(1)
Total	(2n – 2)(8n-8)+1	O(n^2)

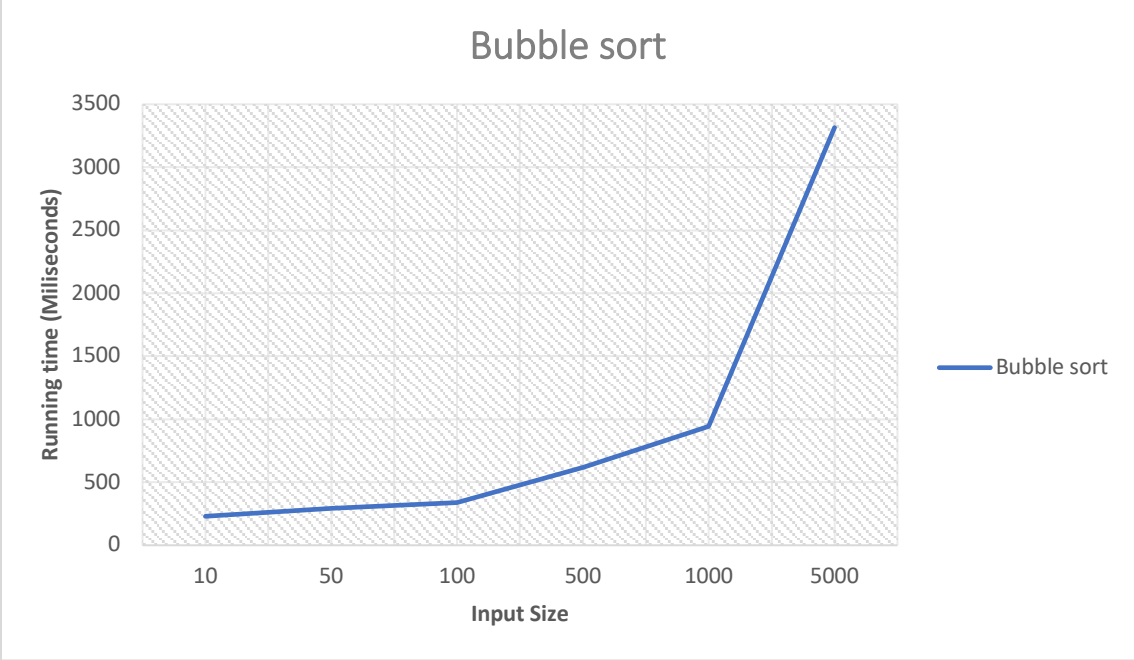
For this algorithm, we will use some of the theorems present in the MDISC notebook together with the values of the table above and all this analysis is related to the worst-case possible complexity.
Like explained before, this algorithm consists of two *for* loops that will iterate over an array. The first one will have n-1 iterations, and for each, in the worst-case scenario there will be made n-1 changes in the second *for* loop iteration. So, the total number of comparations made are also n-1.
We can simplify this with the following expression:

$$\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i = n(n-1) - \frac{1+(n-1)}{2}(n-1) = \frac{n(n-1)}{2}.$$

Now talking about Big-O notation, this algorithm will have O(n^2) as time complexity, this is justified with the same two theorems used in the Big-O analysis of the brute-force:
"$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$, with $a_i \in \mathbb{R}$, for all the i values. So, $f(x)$ is O (x^n)".
"Being f_1 and f_2 functions that $f_1(x)$ is O$(g_1(x))$ and $f_2(x)$ is O $(g_2(x))$. So $f_1(x)f_2(x)$ is O $(g_1(x)\ g_2(x))$".
The first one will justify the fact that for each isolated for loop the complexity and for the comparation process is O (n - 1) and the fact that the summed complexity of the different attributions made is O (1).
The second one will justify why after the multiplication of the complexity of the for loops and the comparation process we will obtain a complexity of O(n^2), since they are nested into each other we multiply their complexity and don't sum them.
To conclude, the complexity of this algorithm is given by O(n) + O(n^2) + O(n^2) + O(n^2) + O(n^2) = O(n^2).

Running time test (different inputs):

In this part we will test our bubble sort algorithm for different sizes of inputs, 10, 50, 100, 500, 1000 and 5000.



Bubble sort conclusion:

With this graphic we can verify that the running time like expected is growing faster for bigger input numbers, leading to a big complexity for huge inputs.

Insertion algorithm:

```
procedure insertionSort (a [n] : CsvInfo)
        for i := 1 to n – 1
                k := a [i]
                j = i – 1
                while j>=0 and a [ j ] > k
                        a [ j+1] := a [ j ]
                        j = j - 1
        a[ j + 1] := k

return a
```

Explanation:

The second chosen algorithm was the insertion sort algorithm.
This algorithm has some similarities with the bubble sort, this one will go through all the members of the array like bubble sort, but it will not swap places if the next member is lower, it will verify if the current member is lower than *n* members behind and until it finds the place where that current member is correctly placed, higher than j – 1 and lower than j + 1, we will be iterating in the while loop and "pushing" the member back in positions. Being in the worst-case scenario all the n members changing positions *n – 1* times, because if they are all in the wrong order for the *n – 1* iterations of the *for* loop, the *while* cycle will act like it is a nested *for* loop.

Time complexity:

For our time complexity we will use the notation used in the MDISC notebook. (A) attribution, (I) increment, (C) comparation, (Op) Operation and (R) return.

Lines	Insertion sort	Big-O
1st	(n-1)A or I + (n-1)C	O(n)
2nd	(n-1)A	O(n)
3rd	(n-1)Op	O(1)
4th	2(n-1)C	O(n)
5th	(n-1)A	O(n)
6th	(n-1)A	O(n)
7th	(n-1)A	O(n)
8th	1R	O(1)
Total	(5n-5)(4n-4) + 1	O(n^2)

For this algorithm, we will use some of the theorems present in the MDISC notebook together with the values of the table above and all this analysis is related to the worst-case possible complexity.

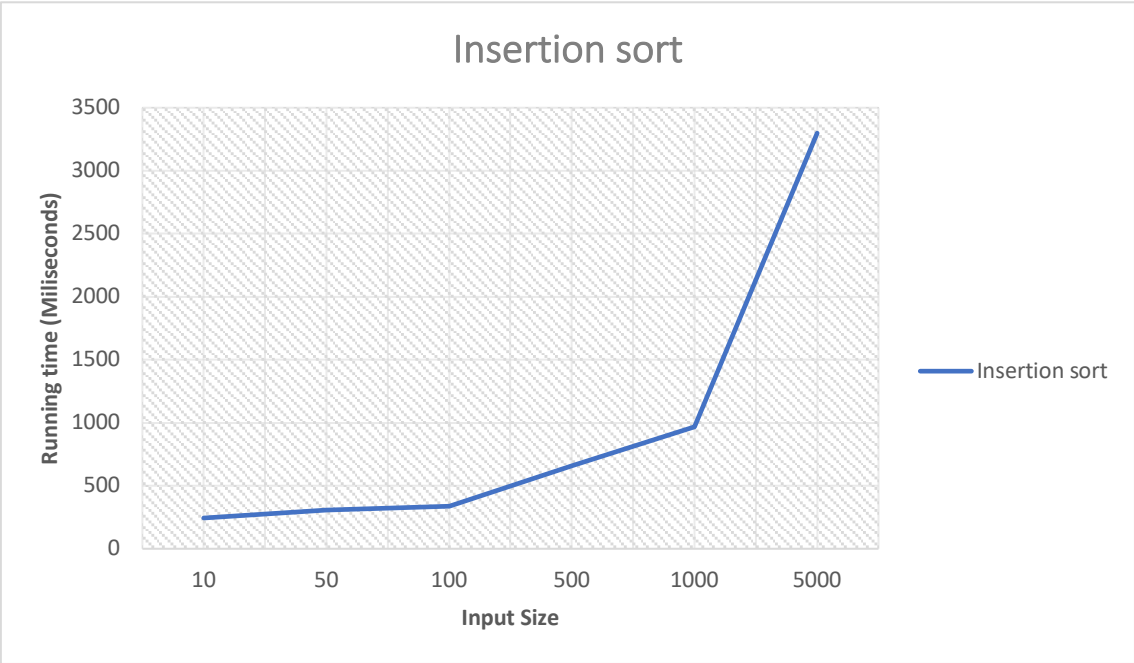
As we said in the worst-case scenario, this algorithm will have pretty much the same logic of the bubble sort, so the time complexity will be similar. In the worst-case scenario like said before, the while cycle will act as a nested for loop, so we can say that the number of swap positions for the n members will be like in the bubble sort, given by this expression:

$$\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i = n(n-1) - \frac{1+(n-1)}{2}(n-1) = \frac{n(n-1)}{2}.$$

Since the logic is the same, the time complexity of this algorithm will be like the bubble sort one and it is given by the sum of O(n) + O(n^2) + O(n^2) + O(n^2) + O(n^2) = O(n^2).

Running time test (different inputs):

In this part we will test our insertion sort algorithm for different sizes of inputs, 10, 50, 100, 500, 1000 and 5000.



Insertion sort conclusion:

With this graphic we can verify that the running time like expected and like the bubble sort algorithm, is growing faster for bigger input numbers, leading to a big complexity for huge inputs. We can see the similarity between both running times in the graphic below.