Advanced Algorithmic and programming

Report TP1

By Folco GALIANO and Antoine JEN

ISEP A2 2016-2017

Minimum of an array

To find the minimum element of an array, you must first look for all the elements of the array. Thus, the complexity is at least *O(n)*.

The algorithm to find the minimum of an array is:

```
public static int minimum(int[] array) {
	int res = array[0];
	for (int i = 0; i != array.length; i++) {
		if (array[i] < res) {
			res = array[i];
		}
	}
	System.out.print("The minimum is: ");
	System.out.println(res);
	return res;
}
```

The algorithm will define the first element of the array as the *res*, thanks to the for loop we run through the array, while browsing the algorithm will compare *res* with the value of the element in the current position, if it’s under the value of *res*, it becomes the new *res*.

Finding the minimum value of an array is fast or even instant for moderate array length.

The complexity of this algorithm is linear since we are having only a for loop thus we multiply the running time of statements inside the loop by the total number of iterations.

To calculate the running time, we use the function executionTime:

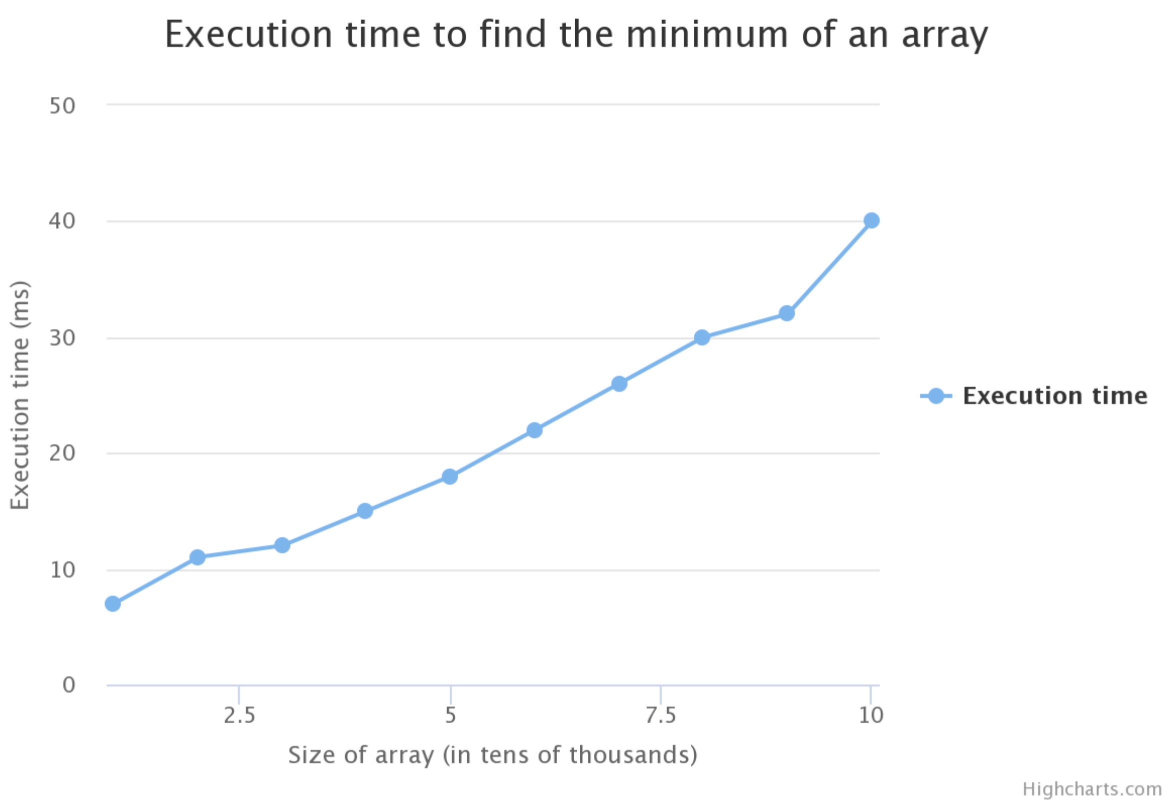
```
public static long executionTime(int[] array) {
	long startTime = System.currentTimeMillis();
	minimum(array);
	long duration = System.currentTimeMillis() - startTime;
	System.out.print("The execution time is: ");
	System.out.println(duration);
	return duration;
}
```

This function defines a start time, then runs the wanted function (here it’s minimum), then defines the estimated time where we subtract the start time to get exactly the time of the running algorithm in ms.

The results are:

```
The execution time is: 7
The execution time is: 11
The execution time is: 12
The execution time is: 15
The execution time is: 18
The execution time is: 22
The execution time is: 26
The execution time is: 30
The execution time is: 32
The execution time is: 40
```

The following chart represents the values shown as output for the calculation of the minimum for different sizes of arrays.



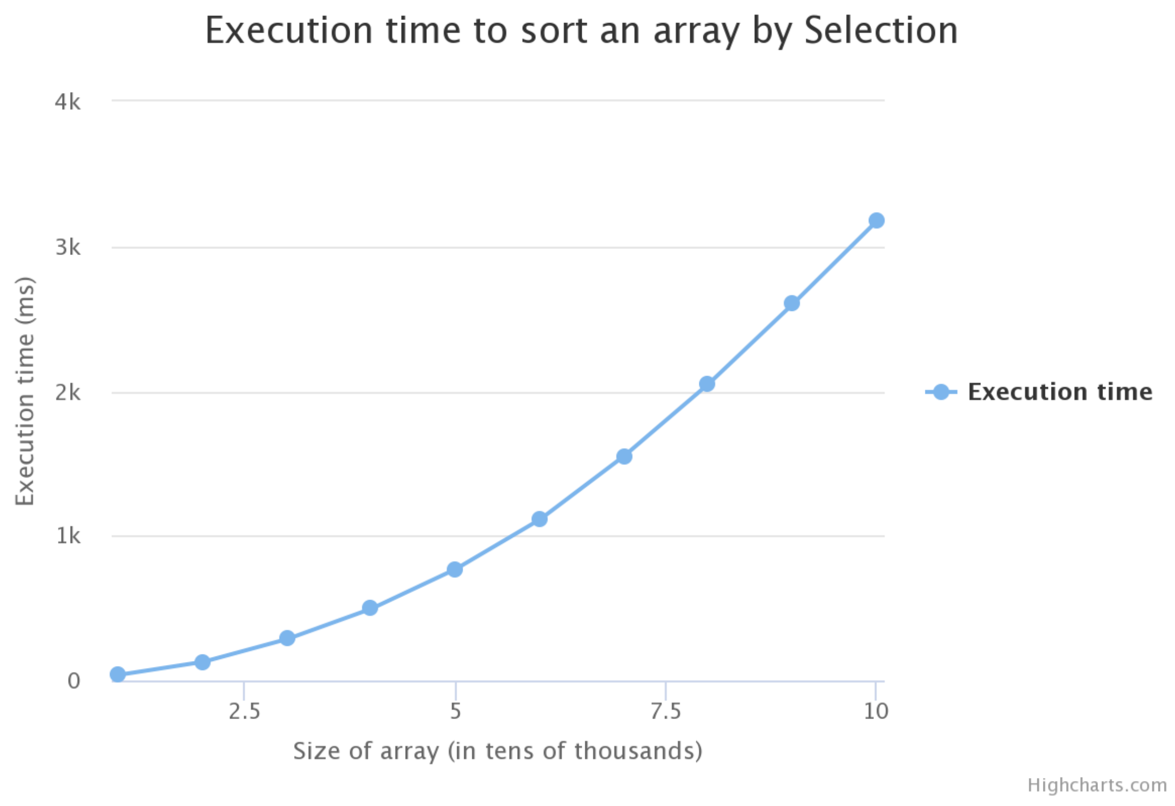
Here, we can suppose that the time complexity of our algorithm is indeed *O(n)*. As expected, the experimental results match the theoretical complexity of our algorithm since the plot is showing a curve having a linear shape. With a greater number of values, we might be able to get a perfect line on our graph.

Sorting algorithms

The complexity of the *swap* function is *O(1)*, since it only does one operation.

Selection sort

```
The execution time is: 39
The execution time is: 129
The execution time is: 287
The execution time is: 497
The execution time is: 771
The execution time is: 1114
The execution time is: 1553
The execution time is: 2050
The execution time is: 2601
The execution time is: 3176
```

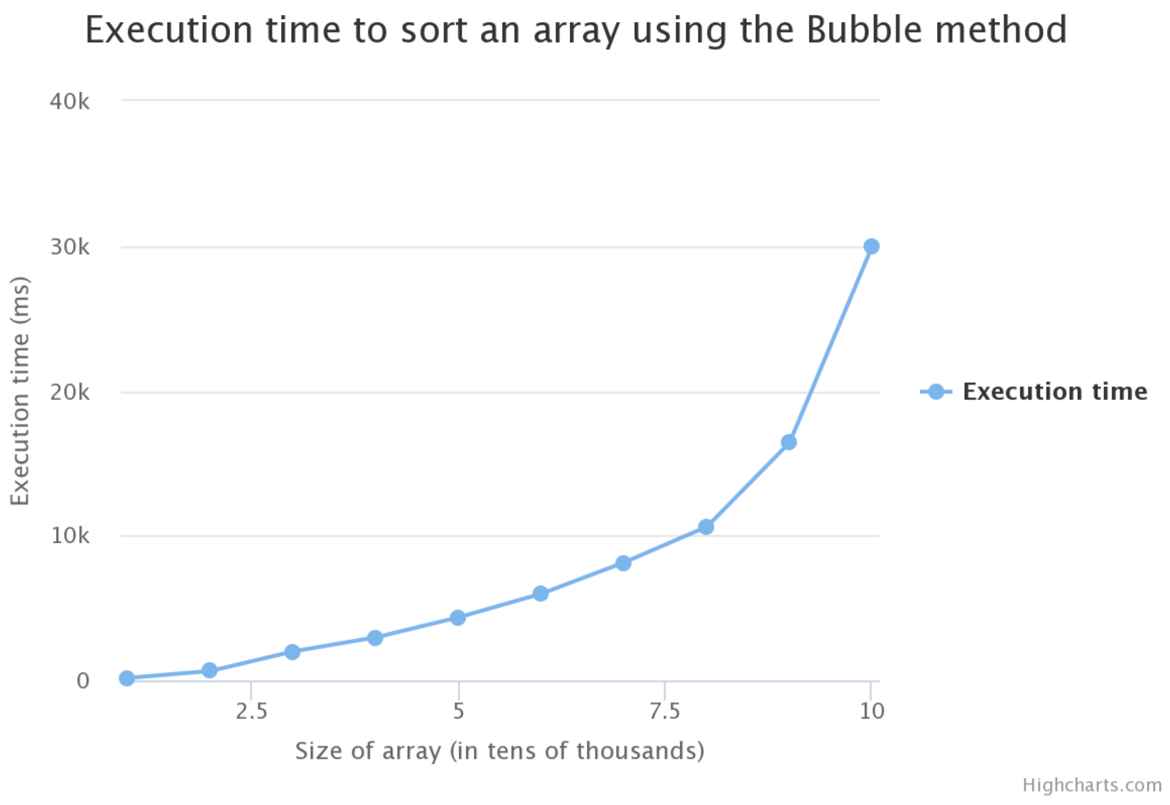


The complexity of the selection sort algorithm is *O(n^2)*, since both `minimumIndex` and `sort` are *O(n)* operations. Thus, the time complexity of our selection sort algorithm will follow a x^2 curve function depending on the number of elements in our arrays.

Bubble sort

Output:

```
The execution time is: 172
The execution time is: 654
The execution time is: 2009
The execution time is: 2977
The execution time is: 4359
The execution time is: 5991
The execution time is: 8150
The execution time is: 10612
The execution time is: 16422
The execution time is: 29974
```



In the code, since we have 2 loops again, we can expect a quadratic complexity, however we are having major difference between the Selection Sort algorithm and the Bubble Sort algorithm, the latter taking much more time.

It can be explained by the fact that the Bubble Sort algorithm requires on average n/4 swaps per entry with each swap involving two entries (Bubble Sort swap elements of the array 2 by 2). Meanwhile, Selection Sort requires only 1 swap since it run through the whole array looking for the minimum then swap it. The complexity of both algorithms is the same however the Bubble Sort algorithm contains more factors which aren't included in the global complexity since they are irrelevant to the n^2 factor.

Hence, by verifying our hypothesis with the previous curve, we can confirm that the complexity of the bubble sort algorithm is $O(n^2)$.

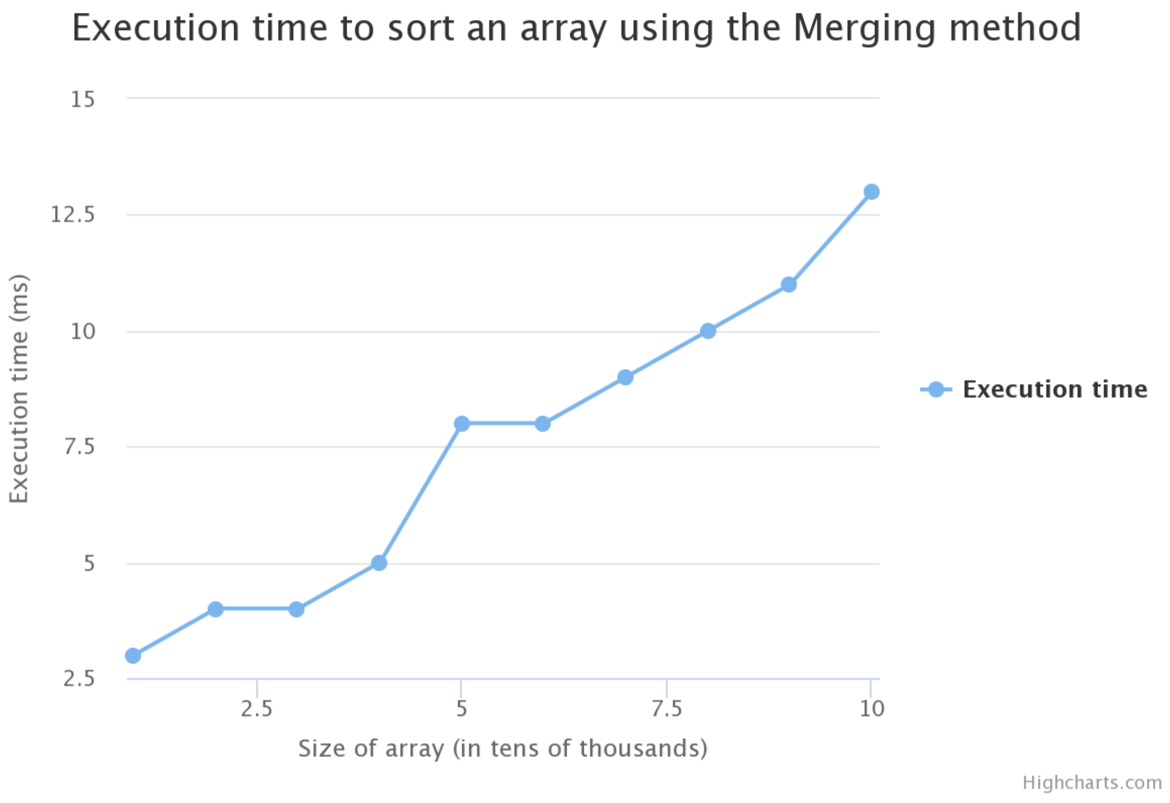
Merge sort

The mergeSorted algorithm is based on dichotomy, which always cuts in half the array where we search the value. Thus, since the size of the array is decreasing over time until the value is found, its complexity converges, hence the *O(log(n))* time complexity.

Moreover, in the algorithm an empty array whose size is half the original is created to merge the two parts, hence the space complexity is *O(n)*.

Output:

```
The execution time is: 3
The execution time is: 4
The execution time is: 4
The execution time is: 5
The execution time is: 8
The execution time is: 8
The execution time is: 9
The execution time is: 10
The execution time is: 11
The execution time is: 13
```



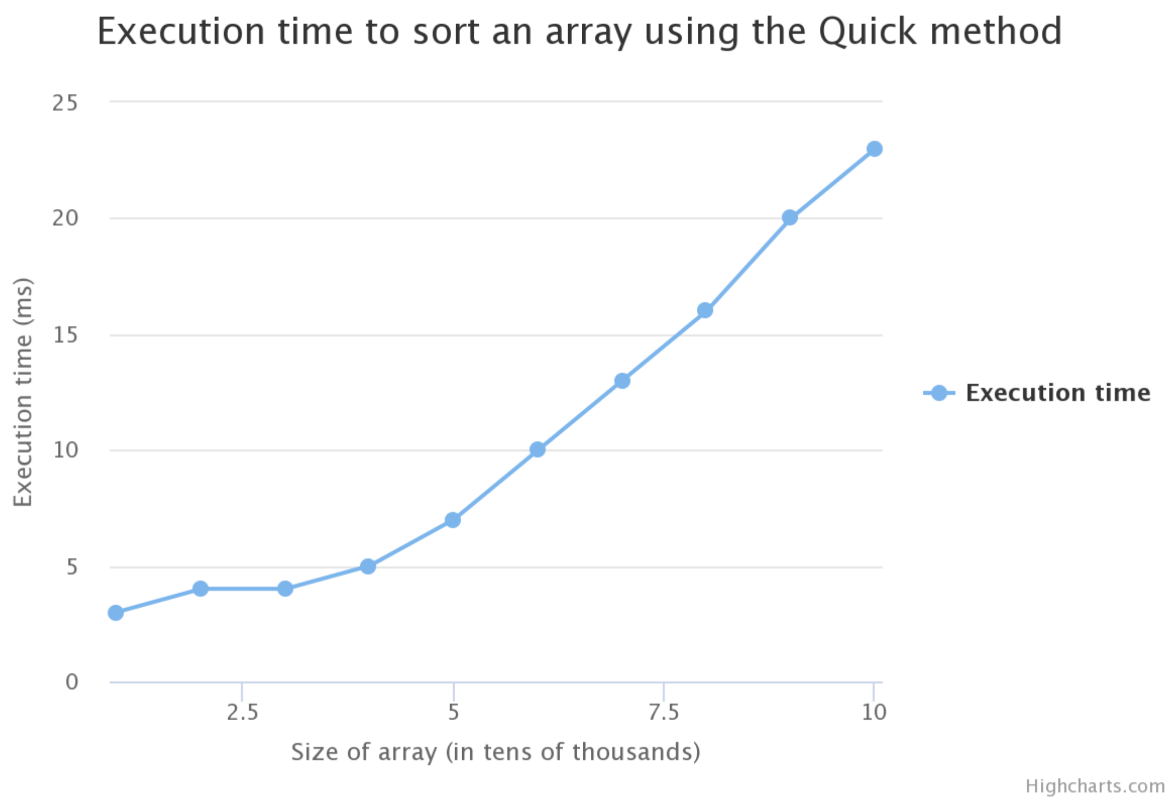
Afterwards, we need to reconstruct the order of the different half-arrays for each iteration, hence the *O(n)* time complexity. Thus, the complexity of the merge sort algorithm is *O(n log(n))*.

Here, if we do not take into account the second value, we can see that the curve is rising slowly then sharply, which looks like an *n.log(n)* function.

Quick Sort

Output:

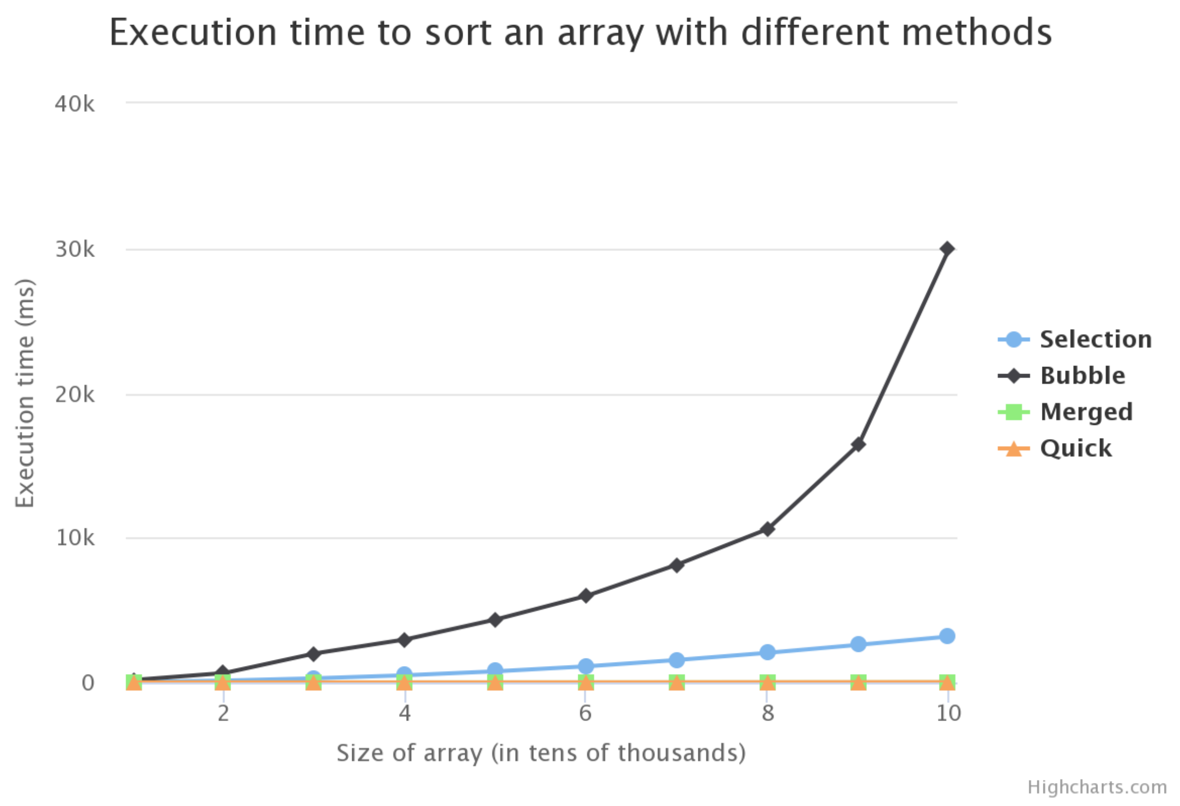
```
The execution time is: 3
The execution time is: 4
The execution time is: 4
The execution time is: 5
The execution time is: 7
The execution time is: 10
The execution time is: 13
The execution time is: 16
The execution time is: 20
The execution time is: 23
```



The complexity of the quick sort algorithm is *O(n.log(n))* normally, and *O(n²)* in the worst-case. In this graph, we can see the curve of the execution time following something between a *n.log(n)* curve and *n²* curve. The pivot selection affects directly the running time of the algorithm: if the pivot is chosen in the way where the subset is very small, it will slow the computation.

Comparison on sorting algorithms

The following chart shows the results of the complexity for the different sorting algorithms used above:



We clearly see that the Bubble Sorting algorithm is the least efficient, while Merged and Quick Sorting, which are both based on “divide and conquer” algorithm design strategy, have the least time complexity. However, while Merged and Quick have the least time complexity, Bubble and Selection have the least space complexity, being *O(1)* and *O(n)*, respectively.

Binary search

The binary search has a *O(log n)* algorithm complexity, and a *O(1)* space complexity, since only the value to be looked for is stored and tested.

With the given algorithm, we are trying to find the number "42" into different sizes of arrays.

Here is the processing time for the different sizes:

Size of array	Output

1 000 000	`21`
	`The execution time is: 9`

5 000 000	`21`
	`The execution time is: 16`

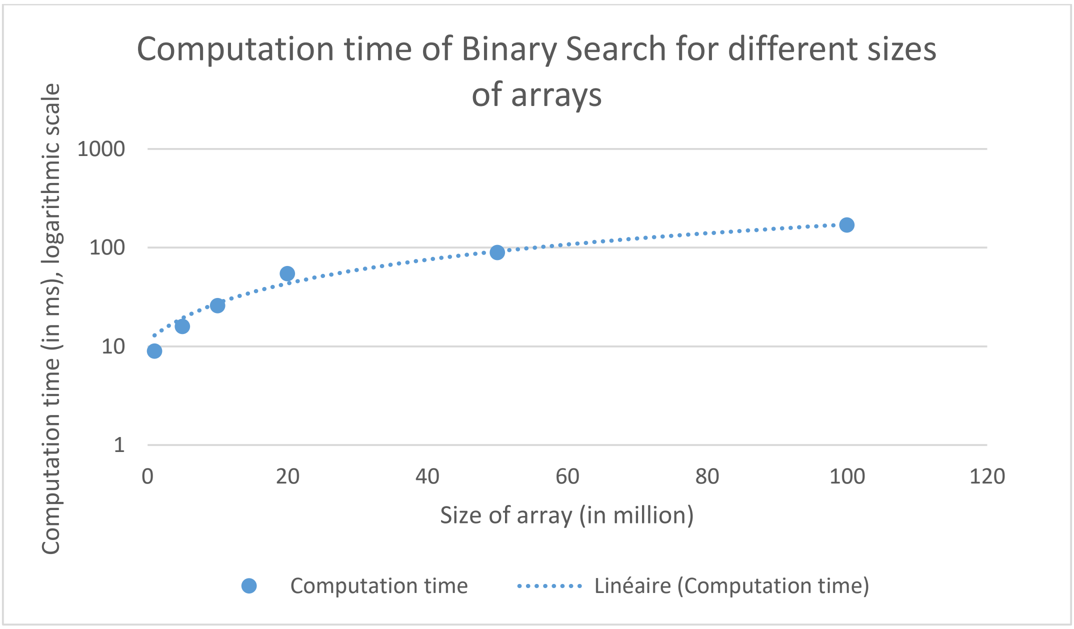
10 000 000	`21`
	`The execution time is: 26`

20 000 000	`21`
	`The execution time is: 55`

50 000 000	`21`
	`The execution time is: 90`

100 000 000	`21`
	`The execution time is: 171`

The values are gathered and displayed in the following graph.


Computation time of Binary Search for different sizes of arrays
Computation time (in ms), logarithmic scale
Size of array (in million)
Computation time
Linéaire (Computation time)