Tutorial course 1: Algorithmic Complexity

Guillaume Lachaud

Contents

List of Figures

The input data

The code of the function *generate1d* was changed because the values generated were not in the desired range.

Minimum of an array

Without knowing anything about the array, we have to look at all the elements of the array to determine which one is the smallest. So, for an array of size n, it takes $n-1$ comparisons. The complexity of the algorithm is thus $\boxed{\Theta(n)}$
Here is the resulting graph

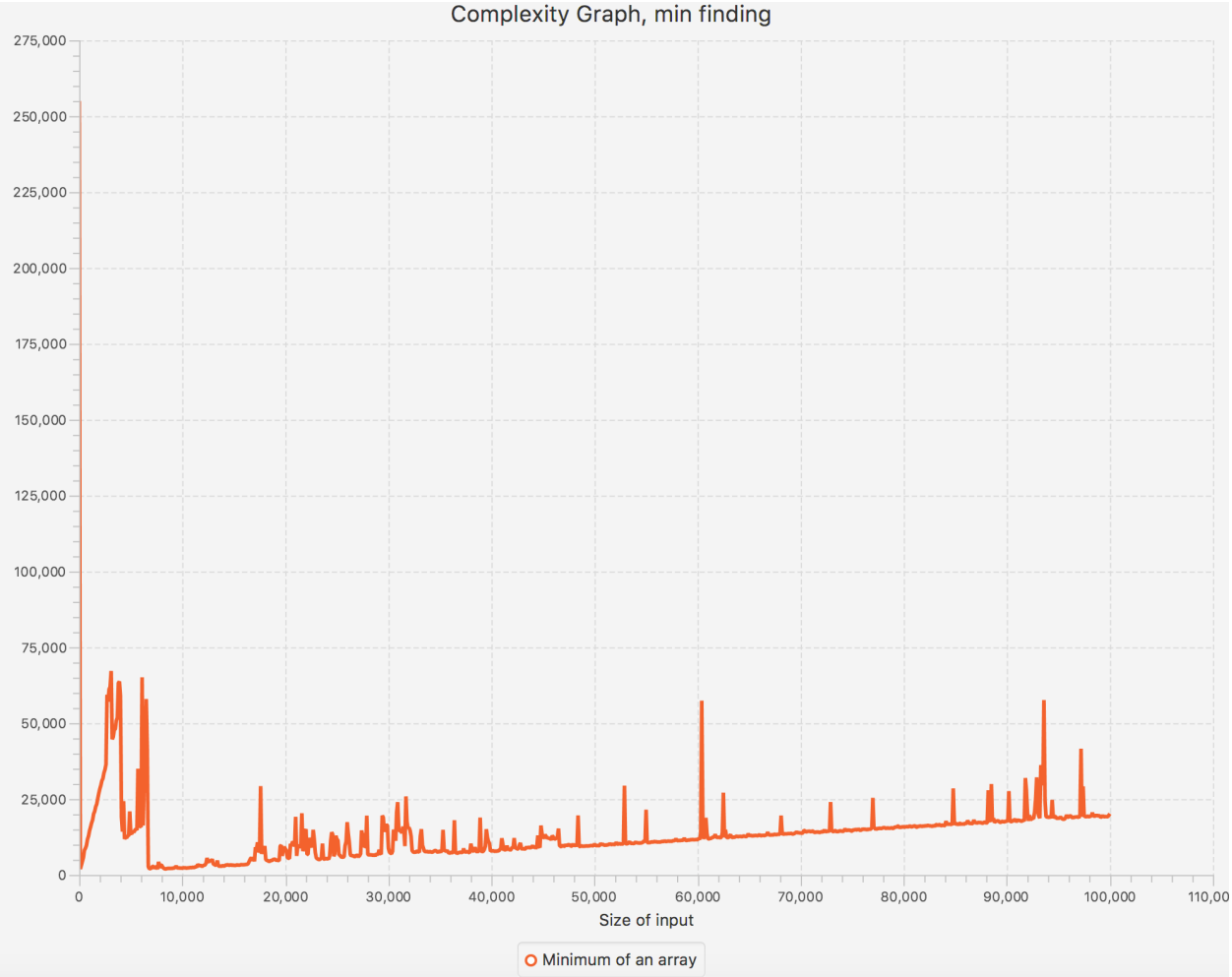

Figure 1: Minimum finding running time (in ns) as a function of the input size

Sorting algorithms

Selection sort

Suppose we have an array of size n. When using the selection sort, we search the array for the smallest element and we swap it with the first element. Then, we search the array minus the first element for the second smallest element, and so on. As we saw in section "Minimum of an array", it takes $n-1$ comparisons to find the smallest element. It then takes $n-2$ comparisons to find the second smallest element, etc. Thus, the total number of comparisons is

$$(n-1)+(n-2)+(n-3)+\cdots+1=\sum_{k=1}^{n-1}k=\frac{n(n-1)}{2}$$

This means the complexity of selection sort is $\boxed{\Theta(n^2)}$

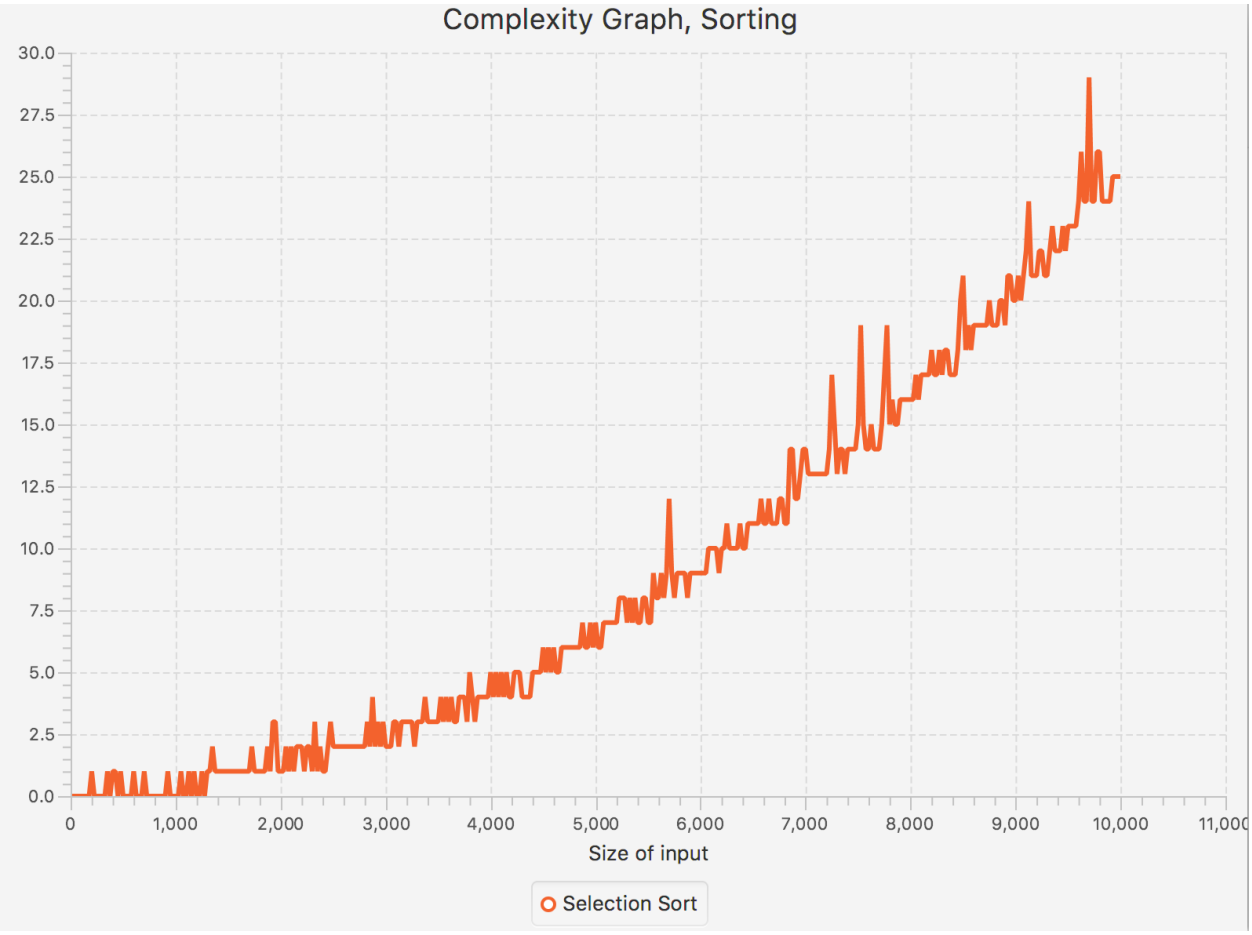

Figure 2: Selection sort running time (in ms) as a function of the input size

When using large enough inputs, the experimental results agree with the theoretical complexity of this algorithm, provided that we multiply the function $x \mapsto x^2$ by a coefficient. The result is the graph below.

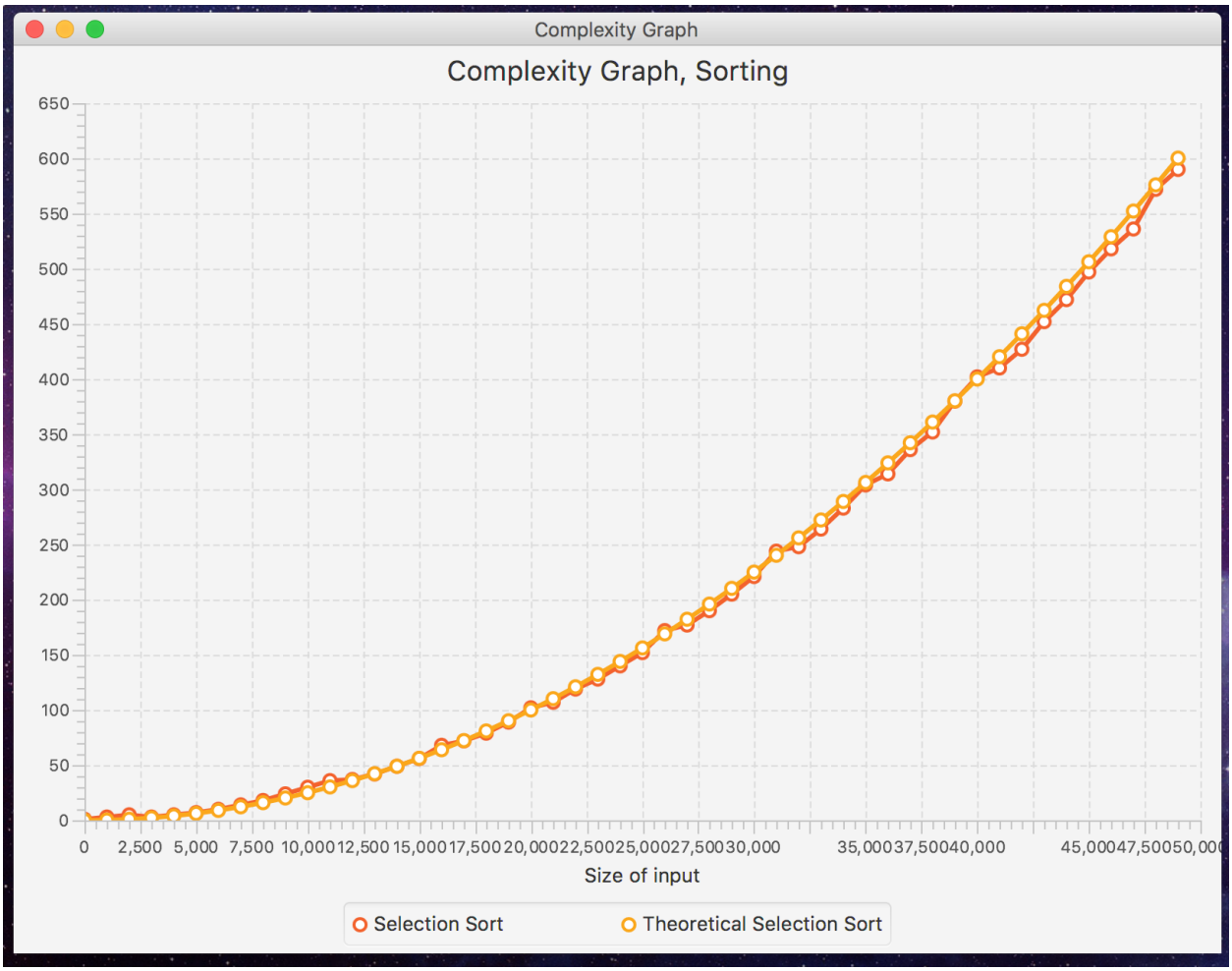

Figure 3: Theoretical and empirical selection sort running time (in ms) as a function of the input size

Bubble sort

The i^{th} iteration of the inner *for* loop puts in i^{th} biggest element in its place. This means that the worst case happens when the array is sorted in the wrong order. In that case, there will be $n-1$ comparisons at each iteration. Thus the total of comparisons will be $n(n-1)$. This means the complexity of bubble sort is $\boxed{O\left(n^2\right)}$

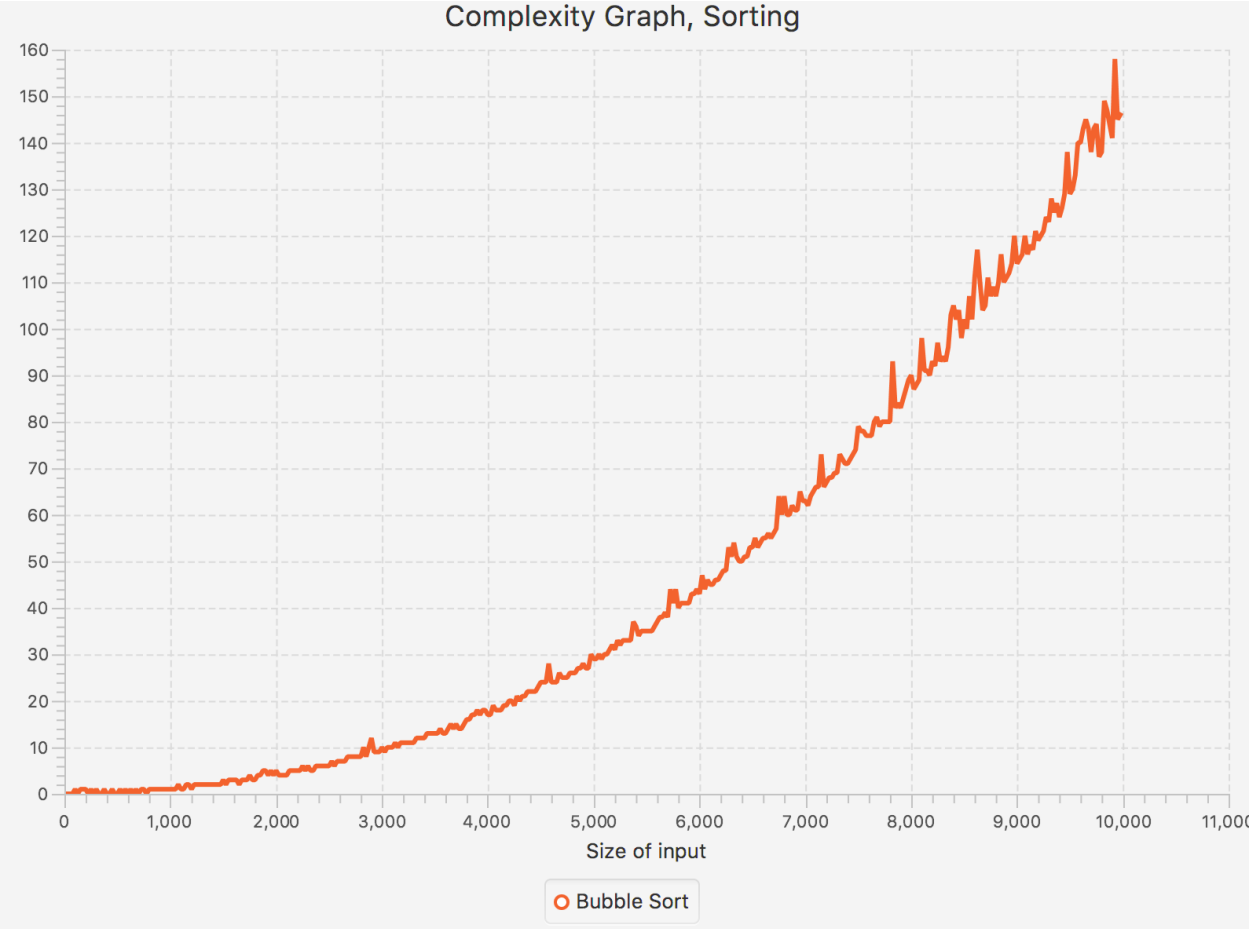

Figure 4: Bubble sort running time (in ms) as a function of the input size

Merge sort

Suppose the first interval has size l and the second interval has size m. There is no inner loop and the main loop iterates through all the values in the first interval and in the second interval. This means there are $l + m$ comparisons. So, the algorithmic complexity of this algorithm is $\boxed{\Theta(l+m)}$ We created a new array of size l in order to do the merging, so the memory complexity is $\boxed{\Theta(l)}$

Given an array such that its size $n = l + m$, the time complexity of merging can be rewritten as $\boxed{\Theta(n)}$ In this case, by denoting T(n) the worst case running time of merge sort, it follows from the implementation that[1]

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \end{cases}$$

with c being a constant. We can then prove that $\boxed{T(n) = \Theta(n \lg n)}$

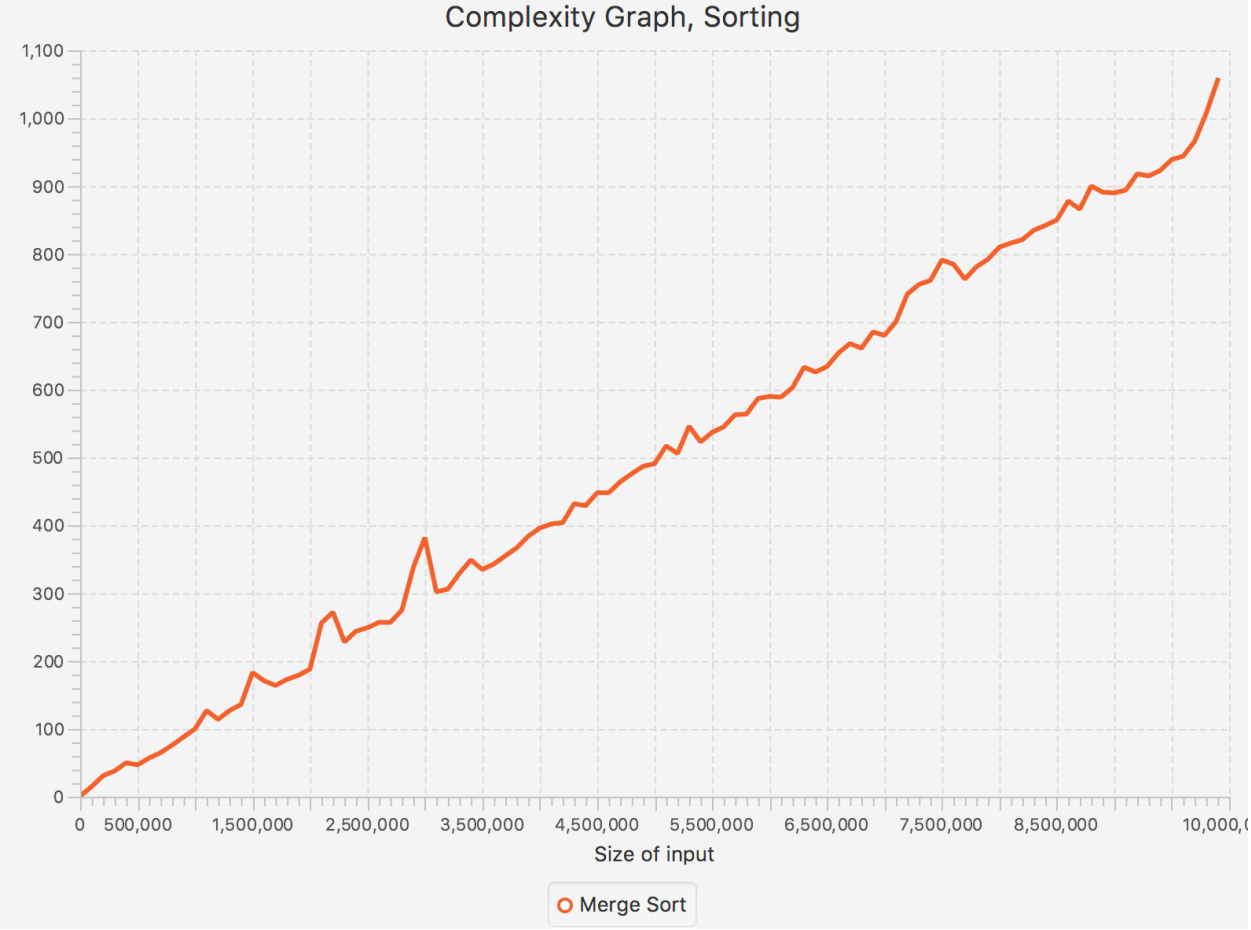

Figure 5: Merge sort running time (in ms) as a function of the input size

[1] We assume that n is a power of 2 but there is no loss of generality.

Quick sort

Given an array of size n, in average, quick sort time complexity is $\boxed{O(n \lg n)}$

In the worst case, its time complexity is $\boxed{O\left(n^2\right)}$ To optimize the quick sort, one way is randomize the input so that worst case inputs can be average if not best case inputs (albeit at the cost of sometimes having a best case input becoming a worst case input for a specific run of the algorithm) or to randomize the choice of the pivot.

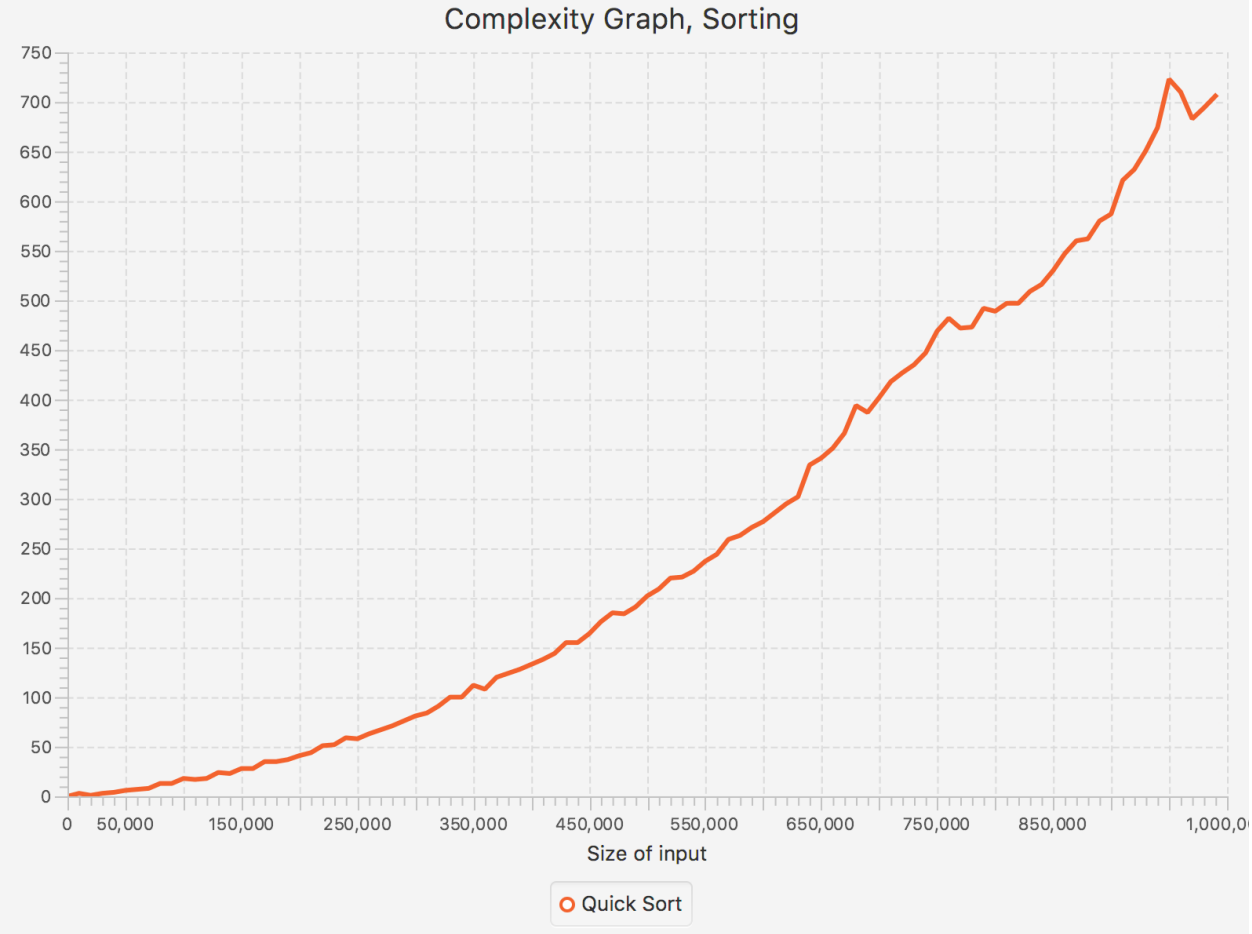

Figure 6: Quick sort running time (in ms) as a function of the input size

Comparison of Sorting algorithms

Between the four algorithms, the best sorting algorithms are *quick sort* and *merge sort*. Although *quick sort* has a worst case time complexity of $O\left(n^2\right)$, it usually runs in $O(n \lg n)$ (n being the size of the array to sort) and, contrary to *merge sort*, it sorts in place, which means no additional memory is required.

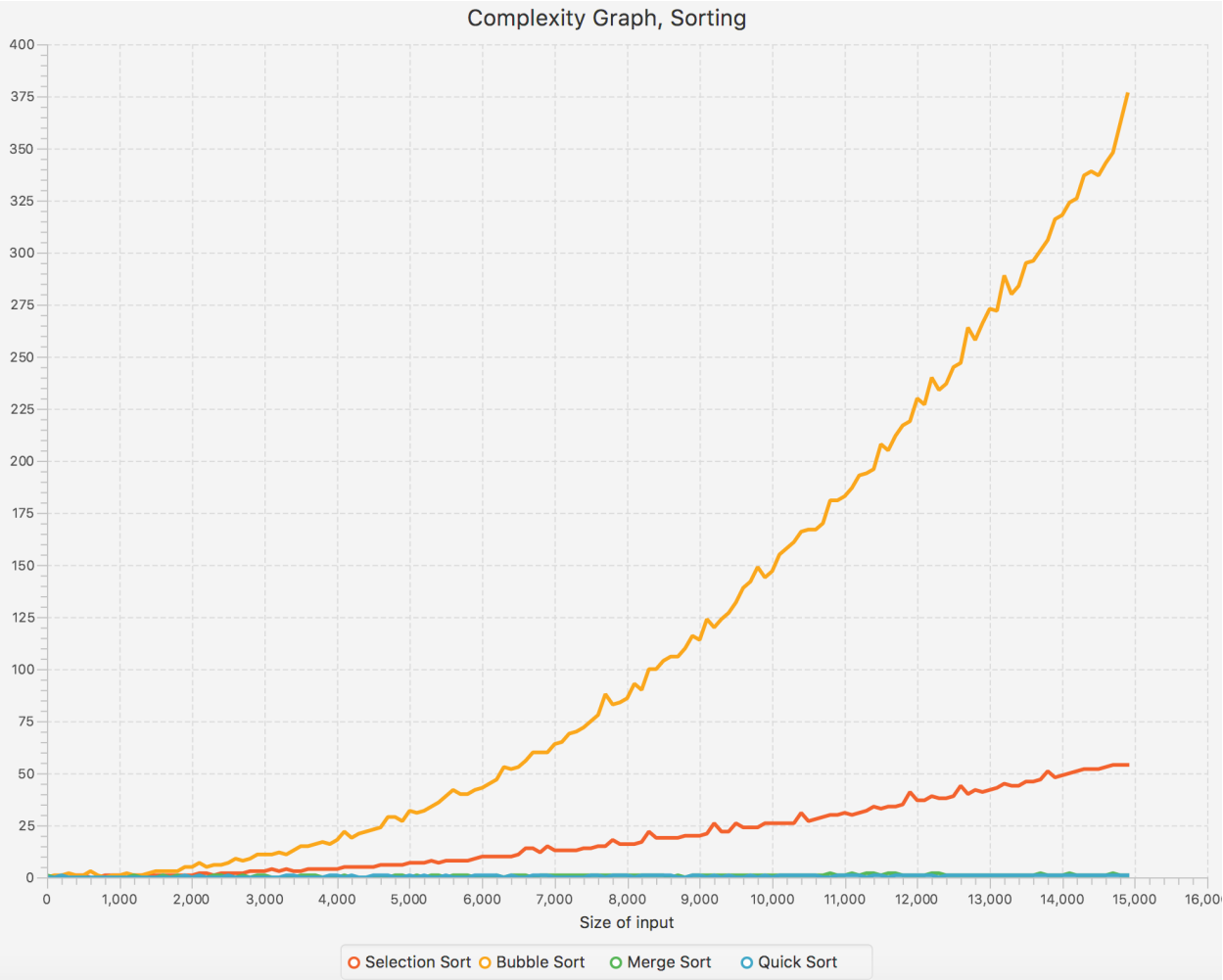

Figure 7: Quick sort running time (in ms) as a function of the input size

Binary search

Given an array of size n, the algorithmic complexity of *binary search* is $\boxed{O(\lg n)}$ (worst case scenario). Its memory complexity is $\boxed{O(1)}$

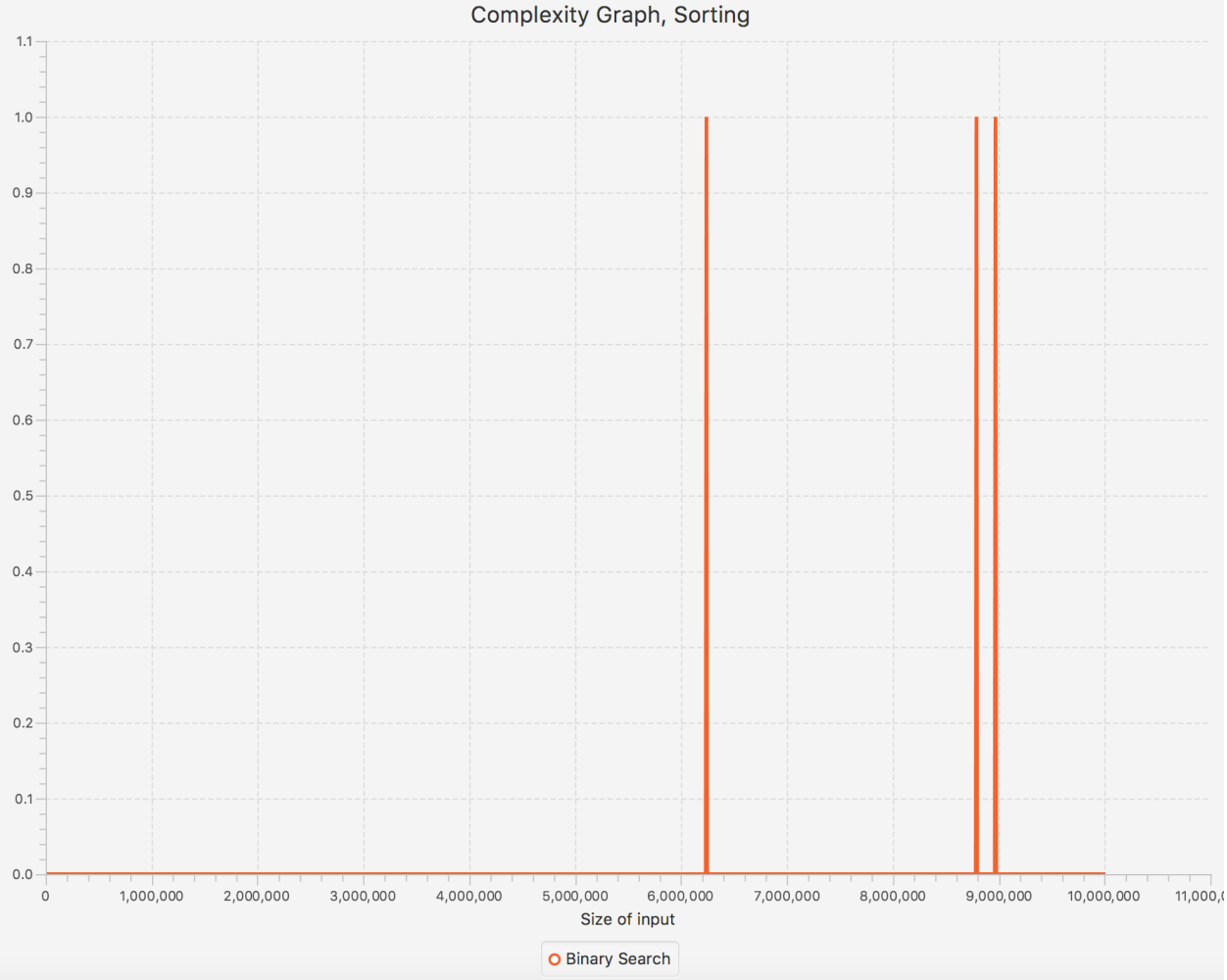

Figure 8: Binary search running time (in ms) as a function of the input size

To have a significant graph, we would need inputs with bigger size (maybe about 1E9 or 1E10 elements). In addition to that, the worst case performance is not always obtained. For it to happen, we need to find the element in the last iteration (when there is only one element left in the array) or when the element isn't in the array (and is smaller than all of the elements of the array, else it would return -1 in the first iteration).

The following graph represents the binary search running time as a function of the input size, but this time, the running time is measured in ns.

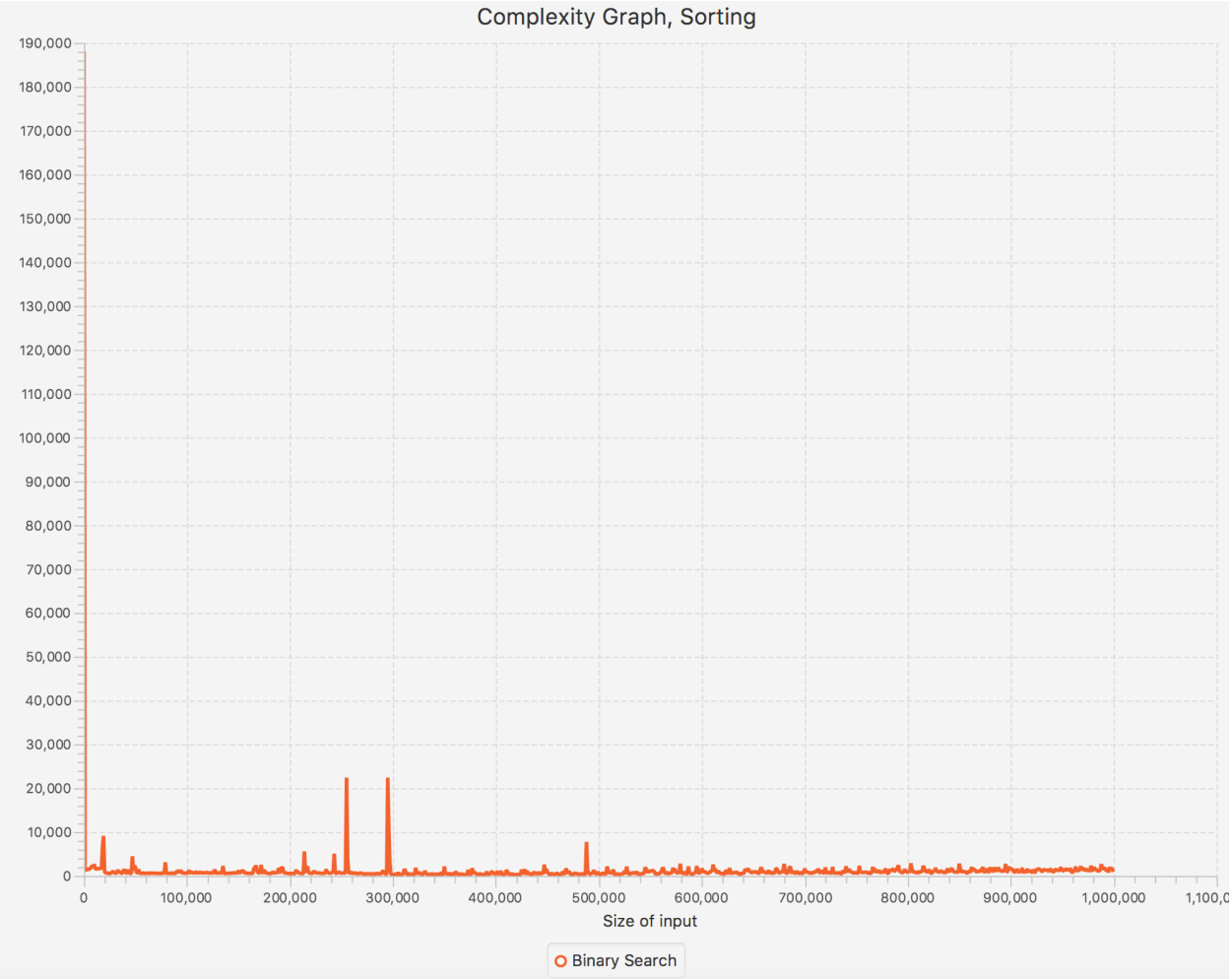

Figure 9: Binary search running time (in ns) as a function of the input size

As we can see, the values are so small that they be considered equal to zero. Furthermore, the calculation of the running time does not represent the actual time spent by the computer doing the binary search, but rather the time between starting the binary search and ending it. Depending on the architecture of the system, it may create some significant differences.