

**ANALYSIS OF ALGORITHMS**

**HOMEWORK 1**

**REPORT**

**STUDENT NAME: ID:**

Yasin Tarakçı 150118055

Yusuf Taha Atalay 150119040

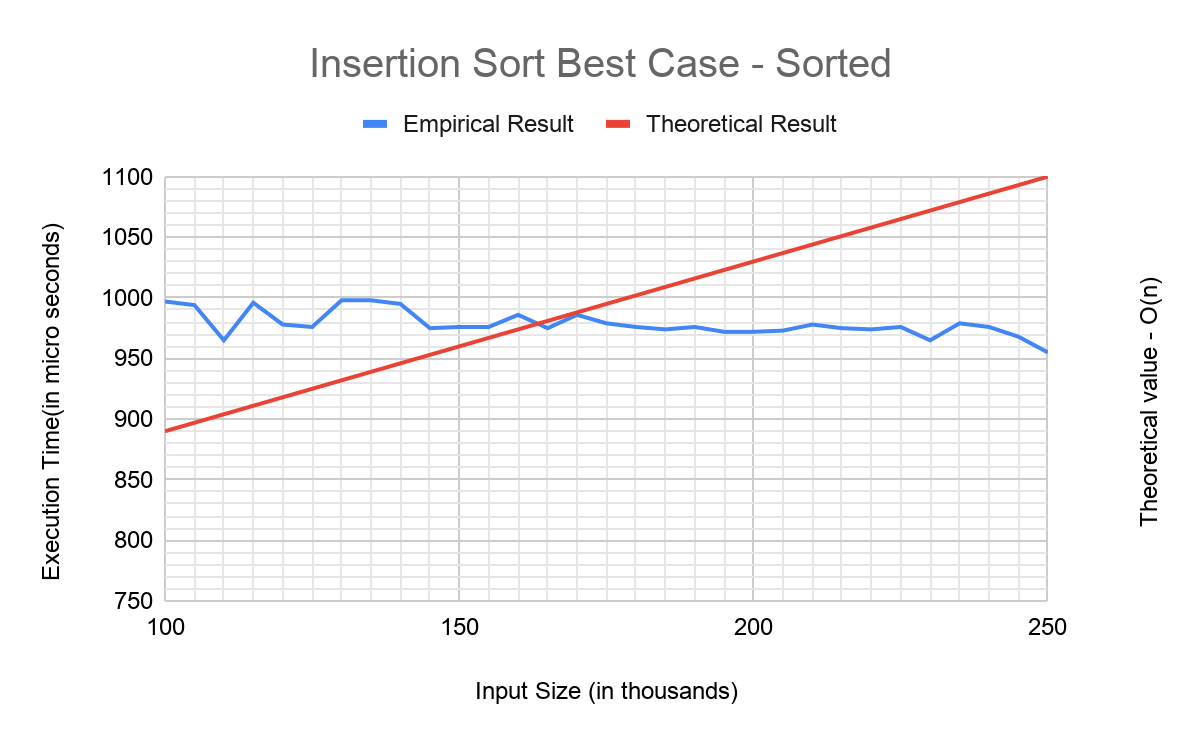
Ahmet Emre Sağcan 150119042

**Submitted To :**  Ömer Korçak

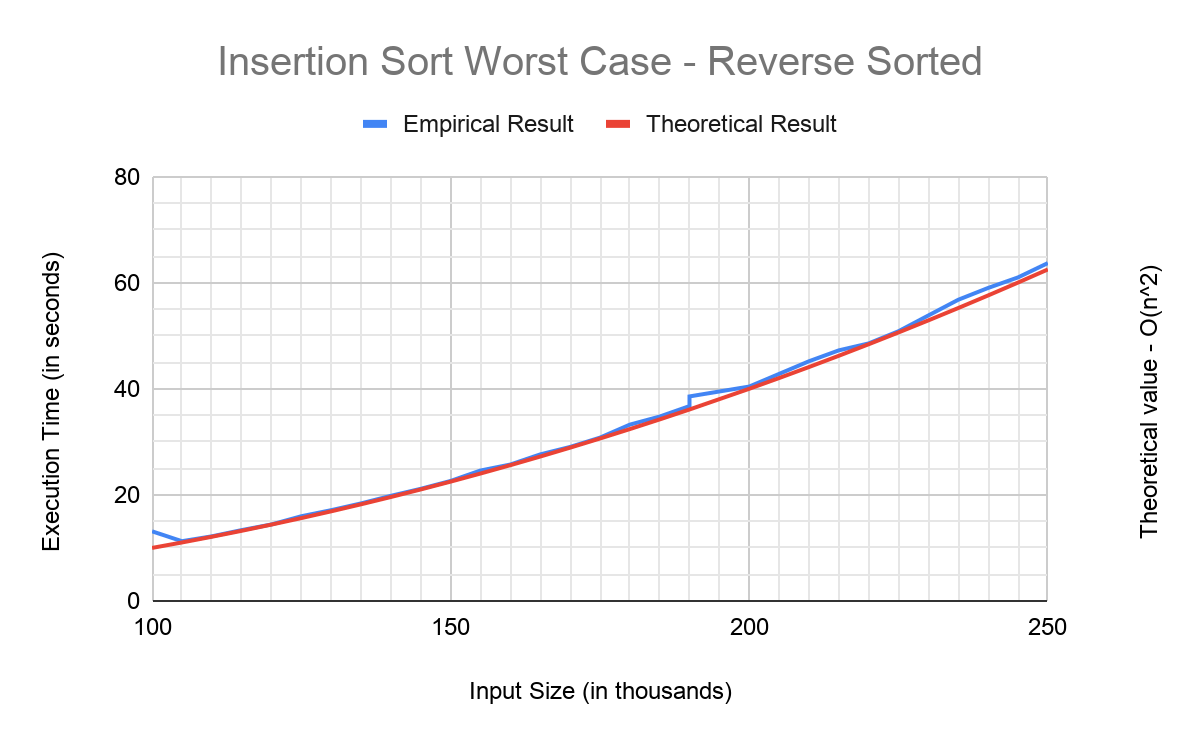
**INSERTION SORT**

**1-Best Case:** Best case for insertion sort is a case in which the array is already sorted. Time complexity for insertion sort for best case is O(n).

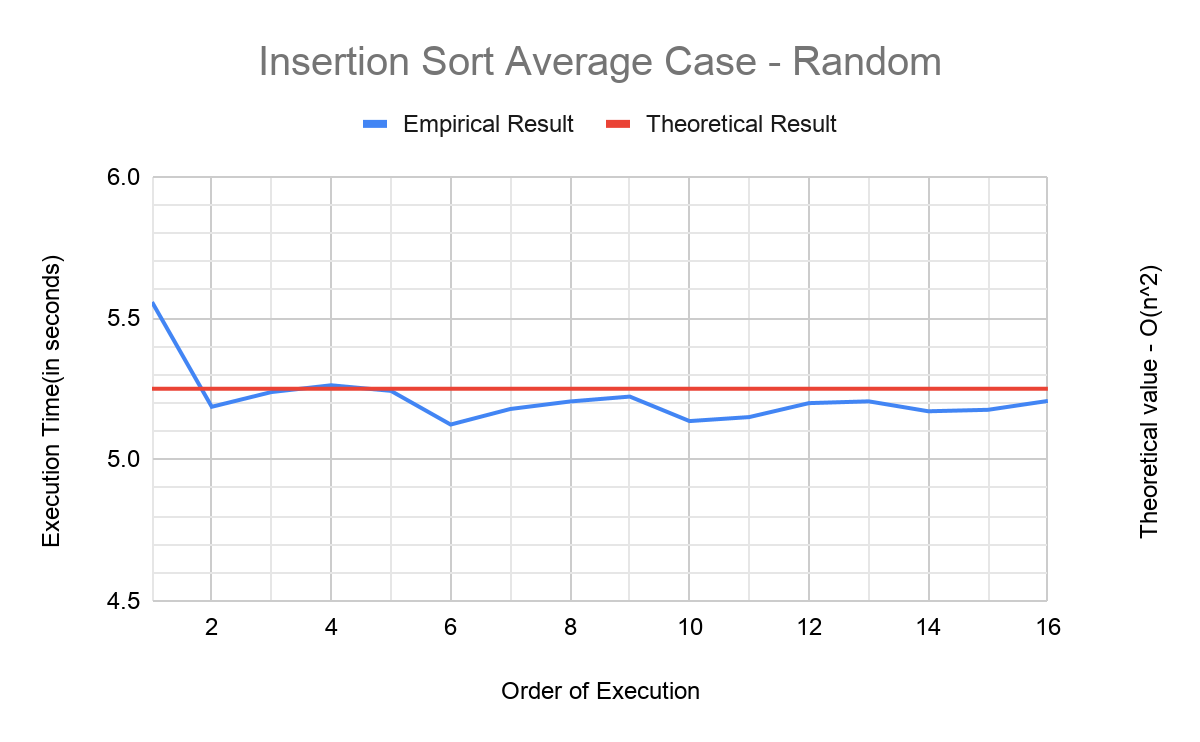
It should be noted that since this sorting happens in very little amount of time and it heavily depends on our computers stability it was hard for us to calculate it precisely. So some of these graphs may look different than their theoretical pairs.



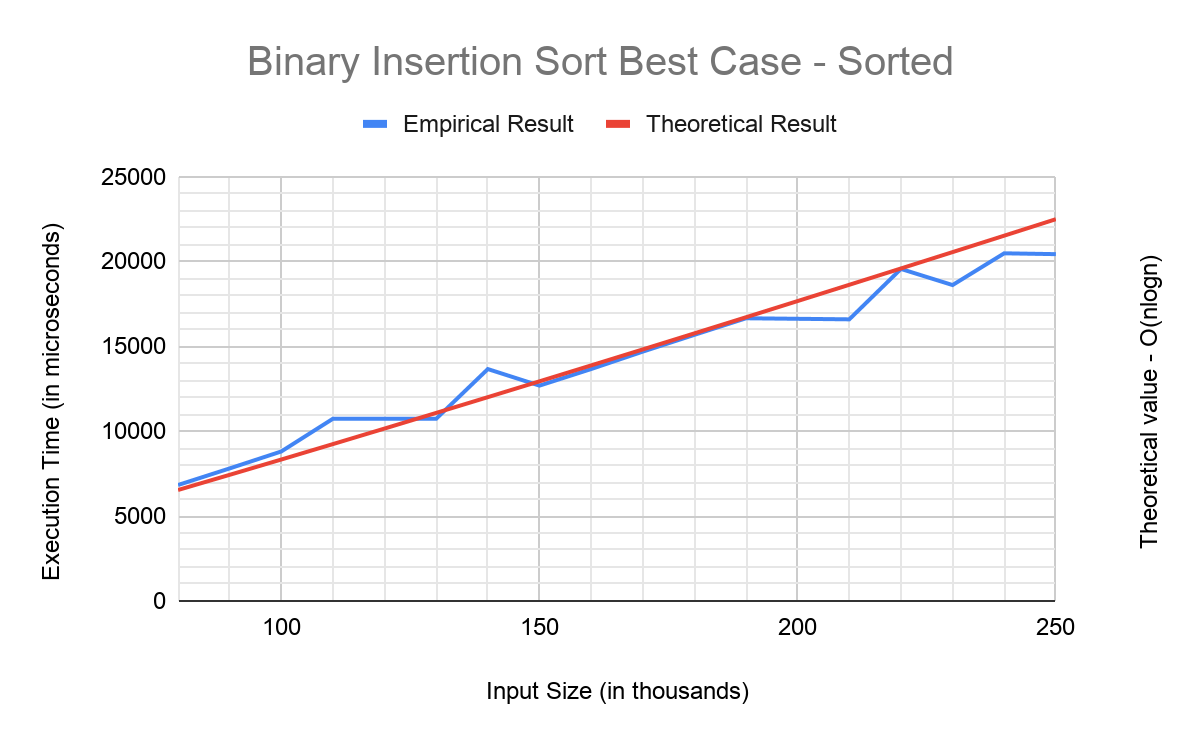
**2-Worst Case**: Worst case scenario for insertion sort is sorting a reverse sorted list. Expected time complexity for this calculation is Ө(). As we can see from the graph that empirical and theoretical graphs are almost identical.



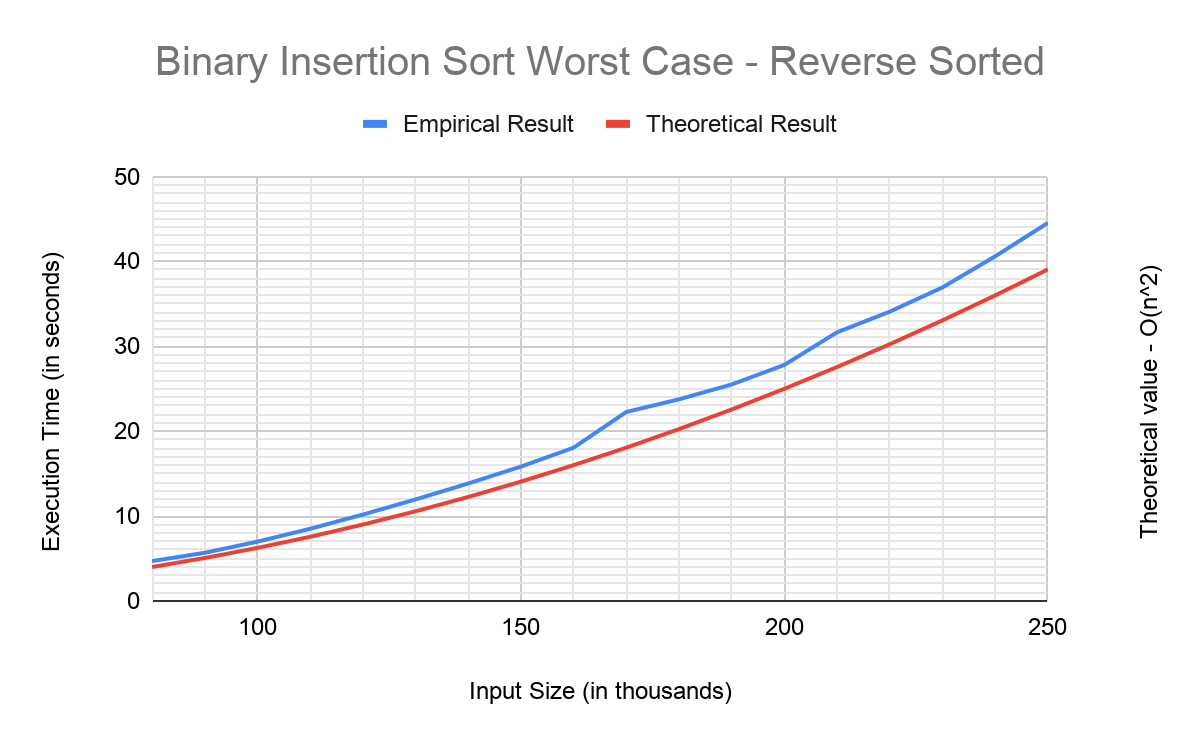
**3-Average Case:** For the average case we make insertion sort calculate randomly generated arrays with the same size (array size = 100000). As we can see from the graph both empirical and theoretical lines are in harmony. Expected time complexity for average case is O().



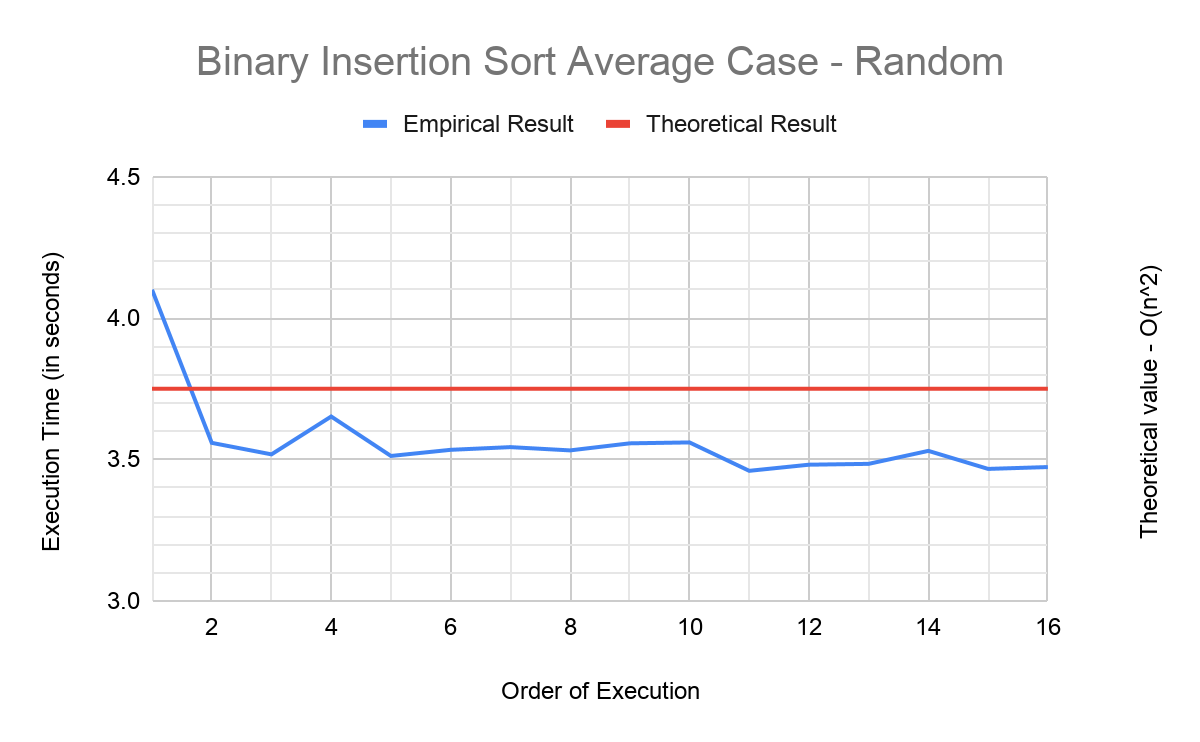
**BINARY INSERTION SORT**

**1-Best Case:** Best case for binary insertion sort is to make it sort an already sorted array. Expected time complexity for O(nlog(n)). If we compare the empirical results with the theoretical graph, both of them have nearly the same lines.

**2-Worst Case:** Worst case for binary insertion sort is to make it sort reverse sorted arrays just like in the insertion sort. Its expected time complexity is O(). We should expect that both of the graphs are parabolical and again some similar values. And as seen in this graph that is the case.



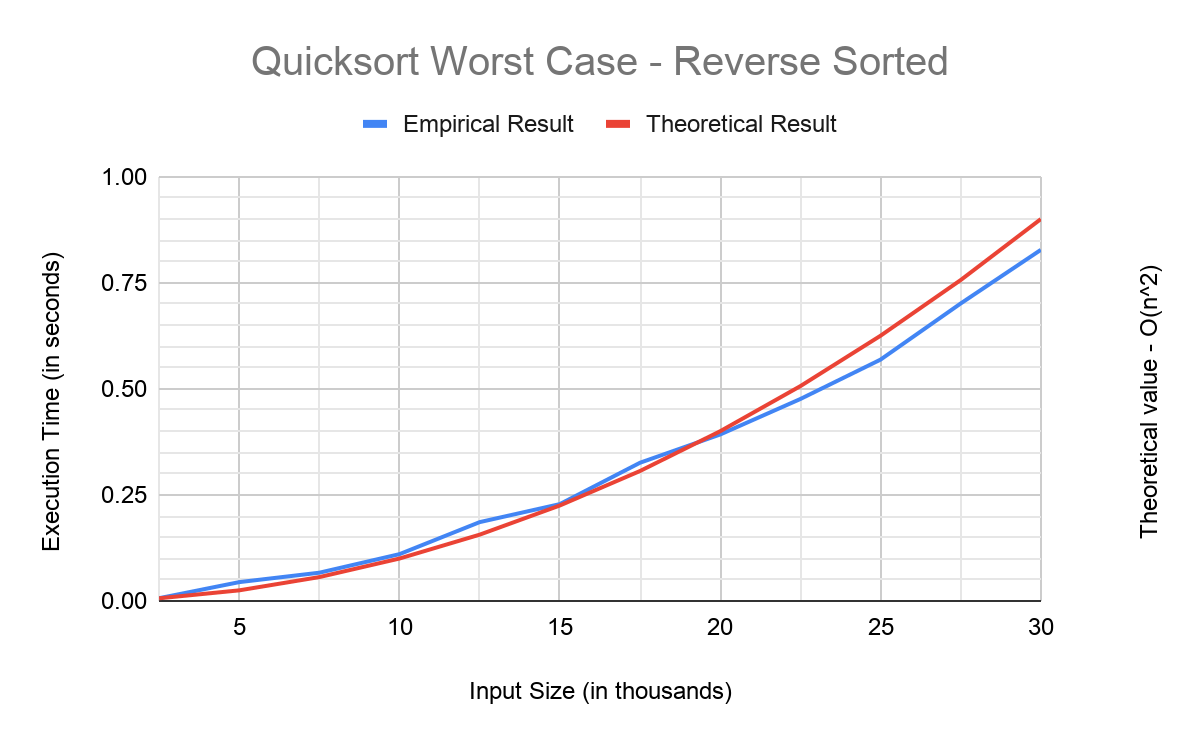
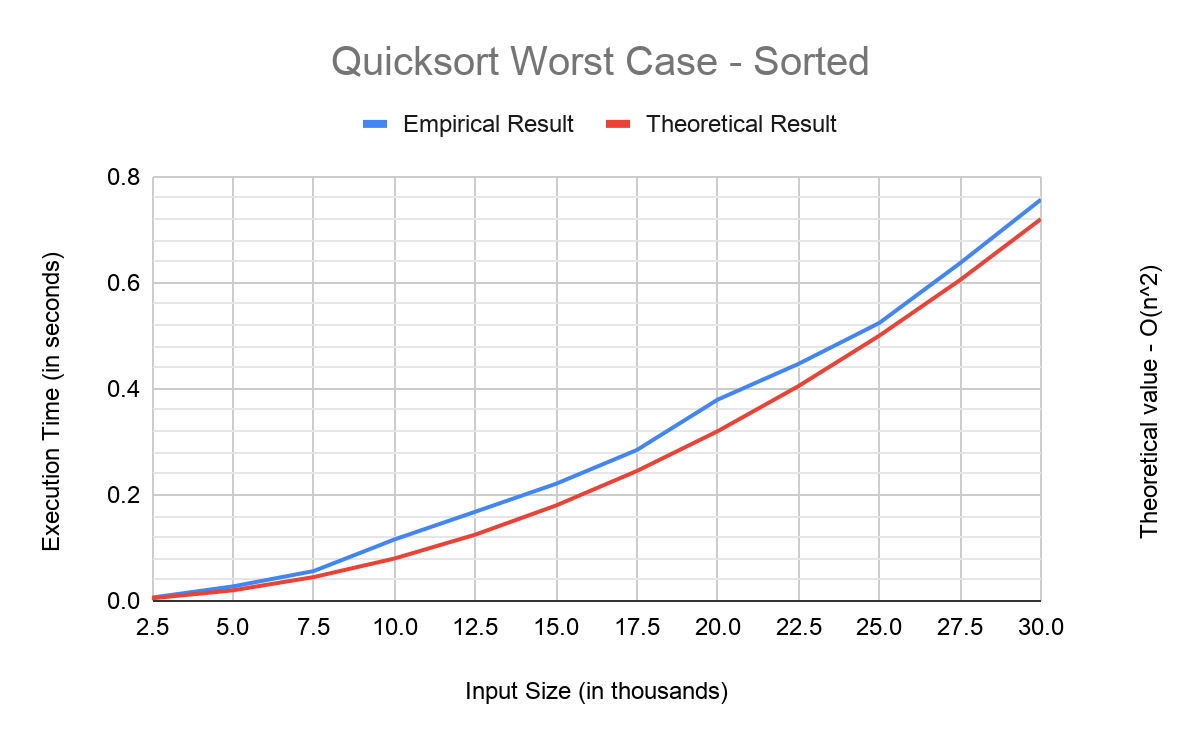
**3-Average-Case:** For the average case we make binary insertion sort calculate randomly generated arrays with the same size (array size=100000). If we look at the empirical results, they have a steady order in average. Time complexity is O().

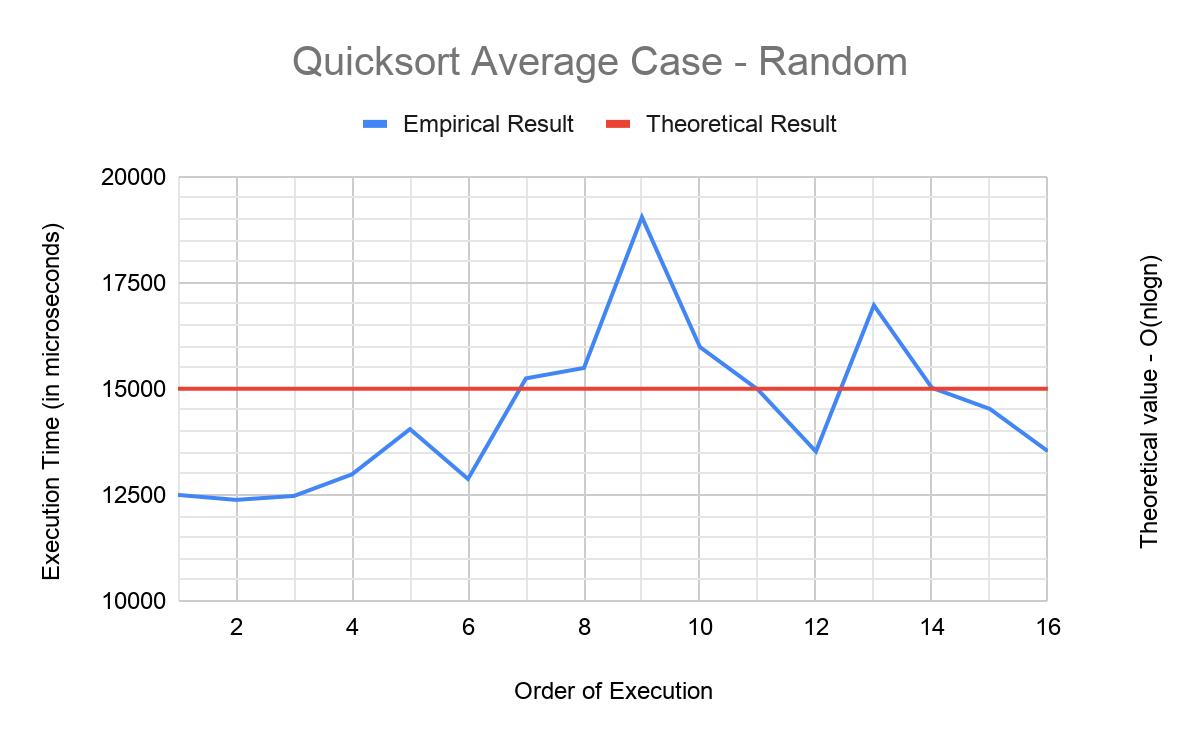


**QUICKSORT**

**1-Best-Case:** Simulating best case scenario for quick sort is very hard to reproduce. So we only consider the theoretical result for this, however we calculate its execution time in numerous cases. Expected time complexity for the best case in quicksort is O(nlog(n)).

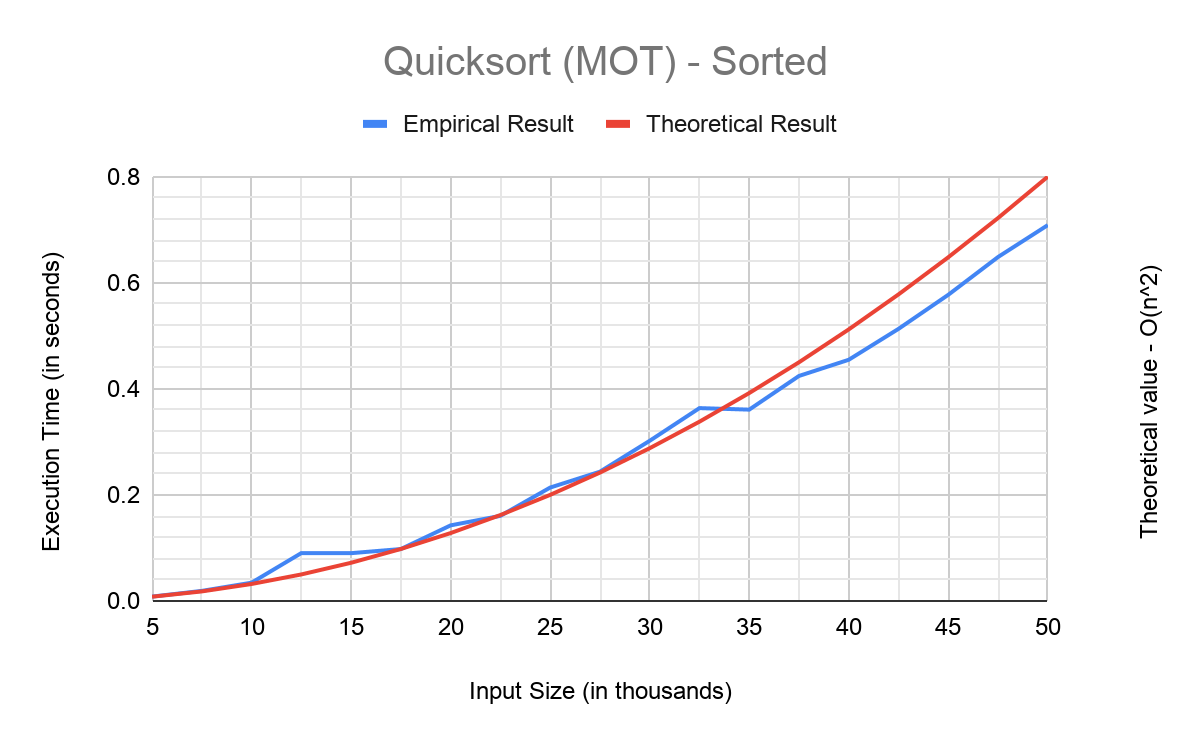
**2-Worst-Case:** Both sorted and reverse sorted arrays create a worst case scenario for quicksort algorithm. Because if the pivot is always the greatest or smallest element it means it’s a worst case scenario and here the first element is our pivot. Worst case time complexity for quicksort is O().

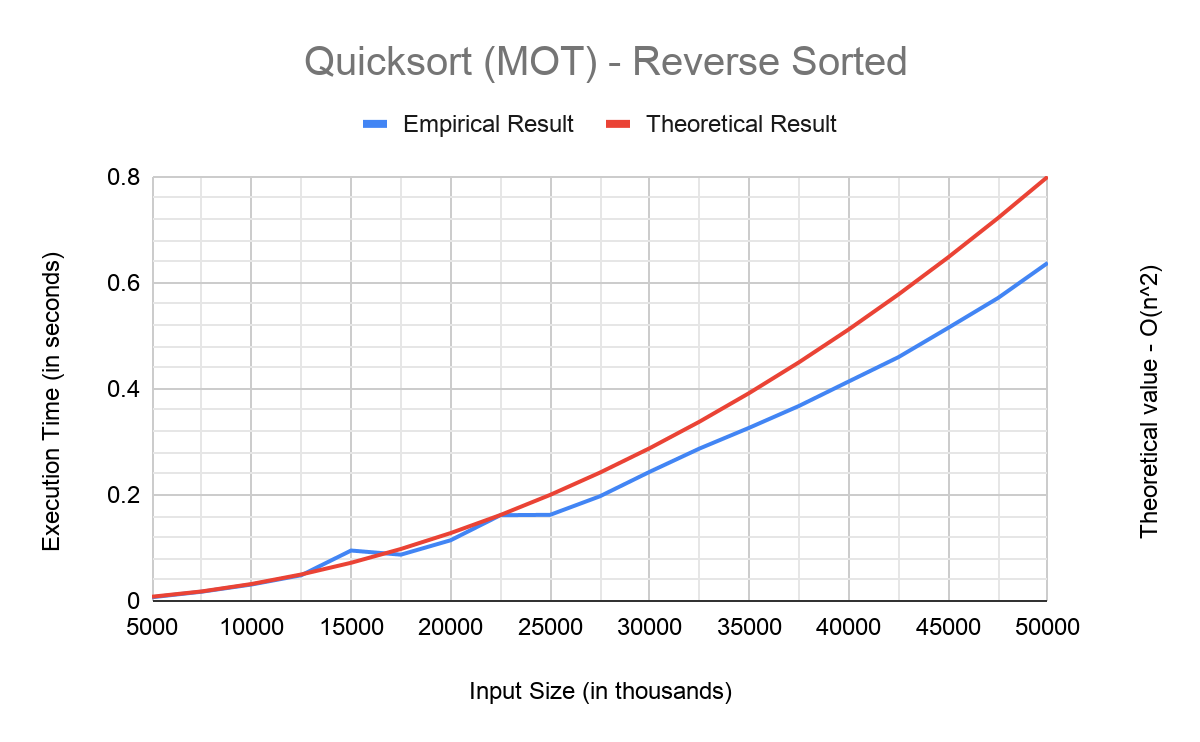


**3-Average-Case:** For the average case we make quicksort calculate randomly generated arrays with the same size (array size = 100000). It happens when the array is not sorted in any direction. Time complexity is O(nlog(n)).

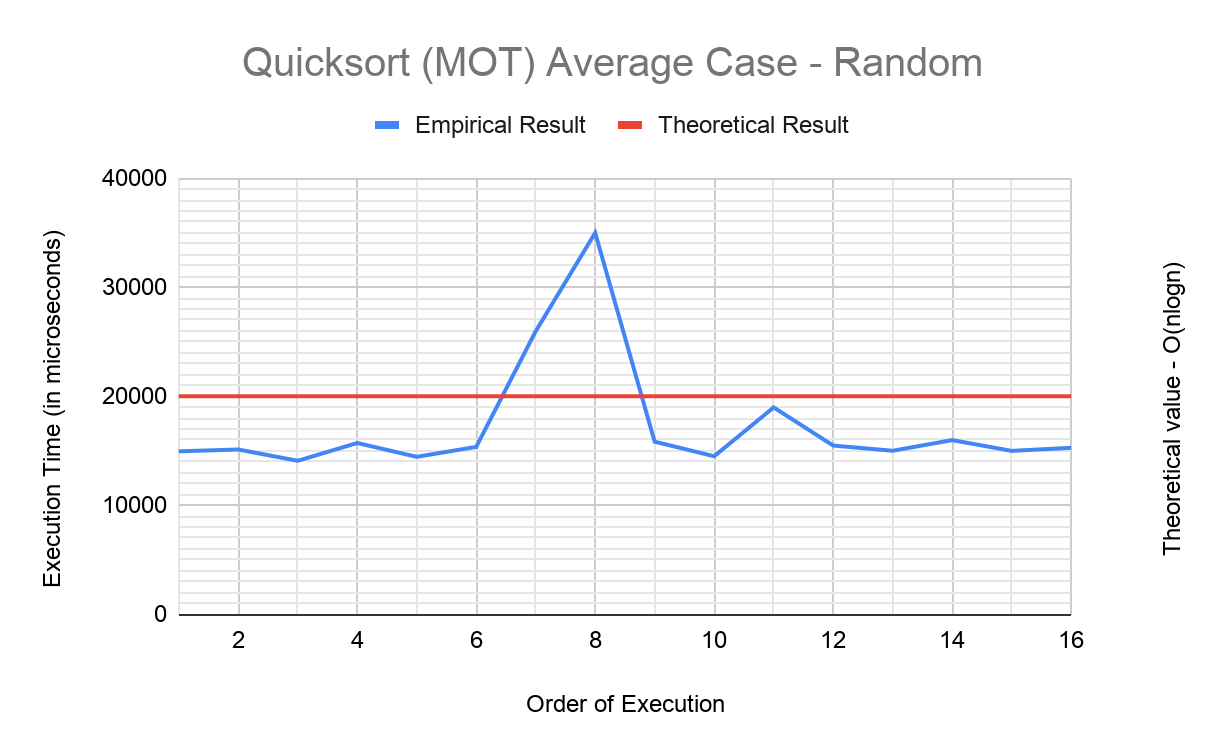
**QUICKSORT (Median of Three)**

**1-Best-Case:** It’s the same with the above quicksort. Simulating best case scenarios for quick sort is very hard to reproduce. Expected time complexity for best case in quick sort is O(nlog(n)).

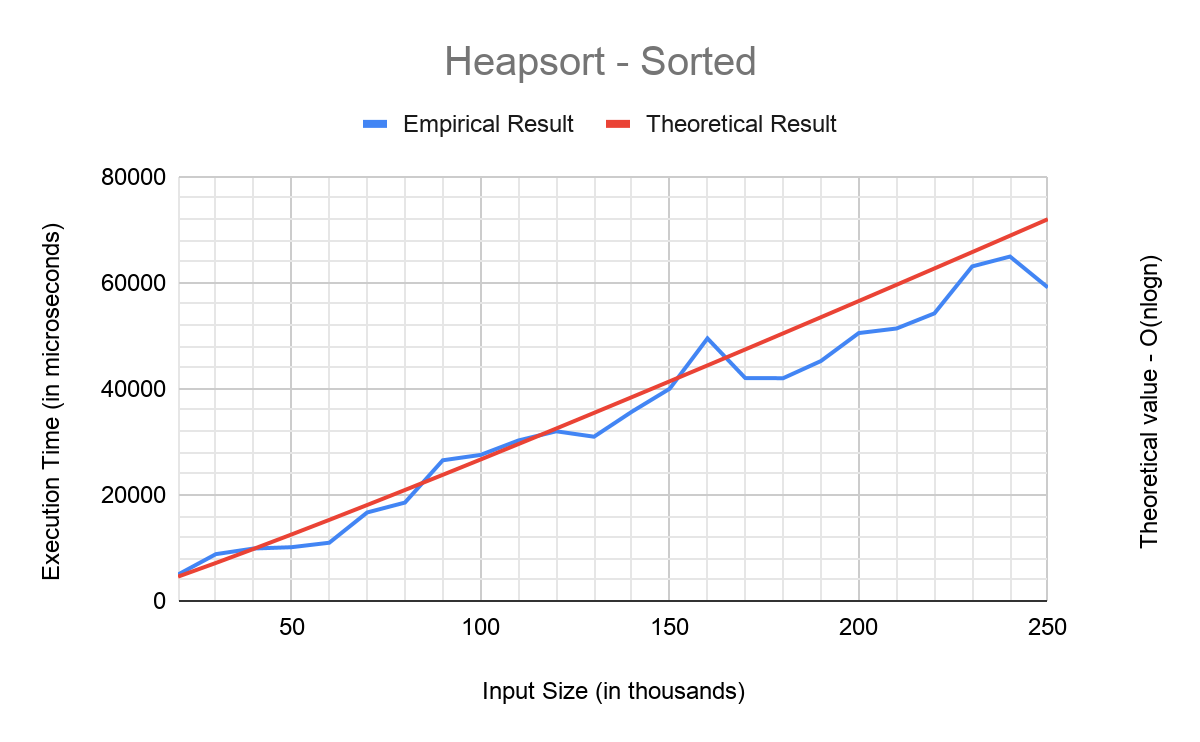
**2-Worst-Case:** In a normal Quicksort algorithm the worst case scenario is where the pivot is the greatest or smallest element in the array. But in this case the “median of three” of an array is the pivot, so there isn’t any certain worst case for that. However when examined in sorted and reverse sorted arrays empirical results were approximately similar with the normal one and theoretical results. Time complexity is O().

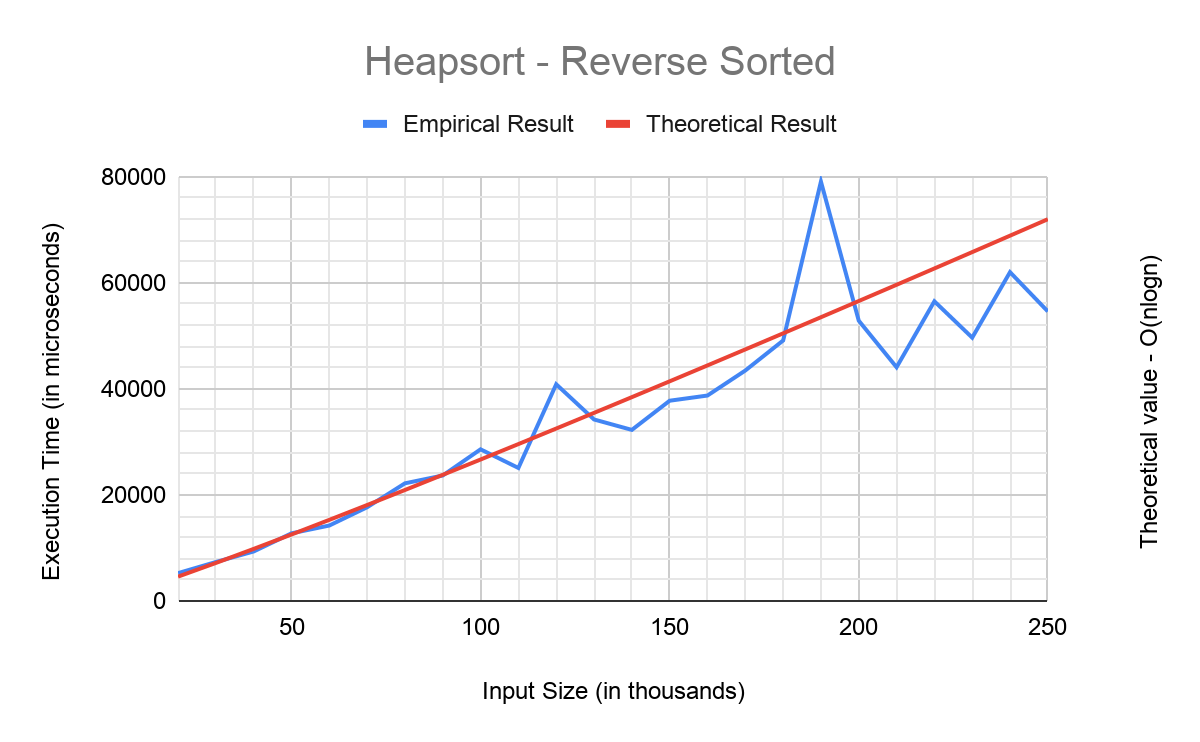


**3-Average-Case:** For the average case we make quicksort (mot) calculate randomly generated arrays with the same size (array size = 100000). Despite the fact that both of the quicksort methods take similar time, the “median of three” method is a bit faster than the regular one. Time complexity is O(nlog(n)).

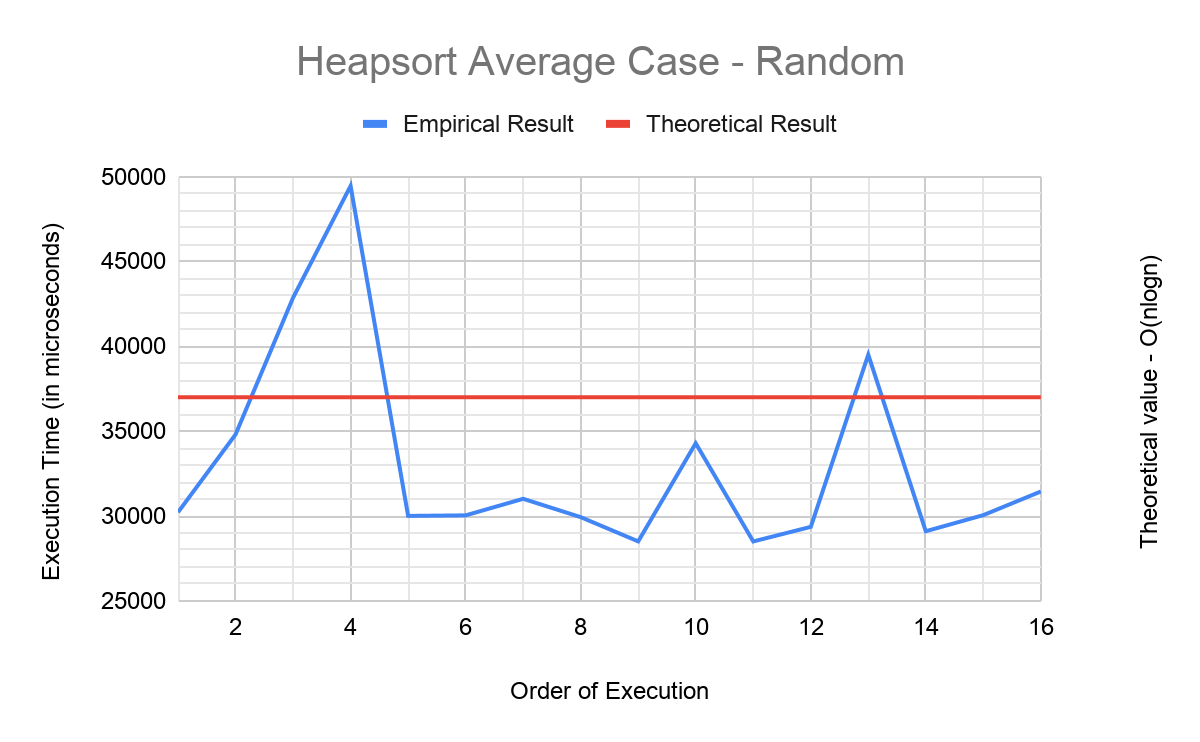


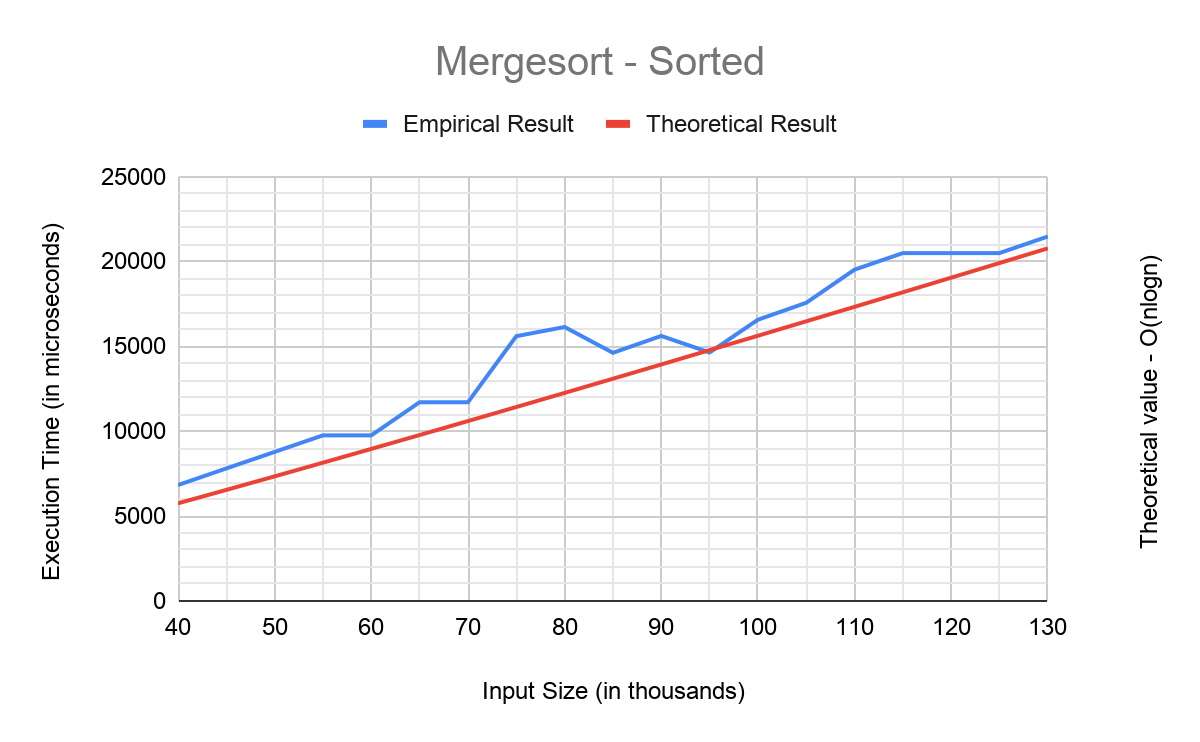
**HEAPSORT**

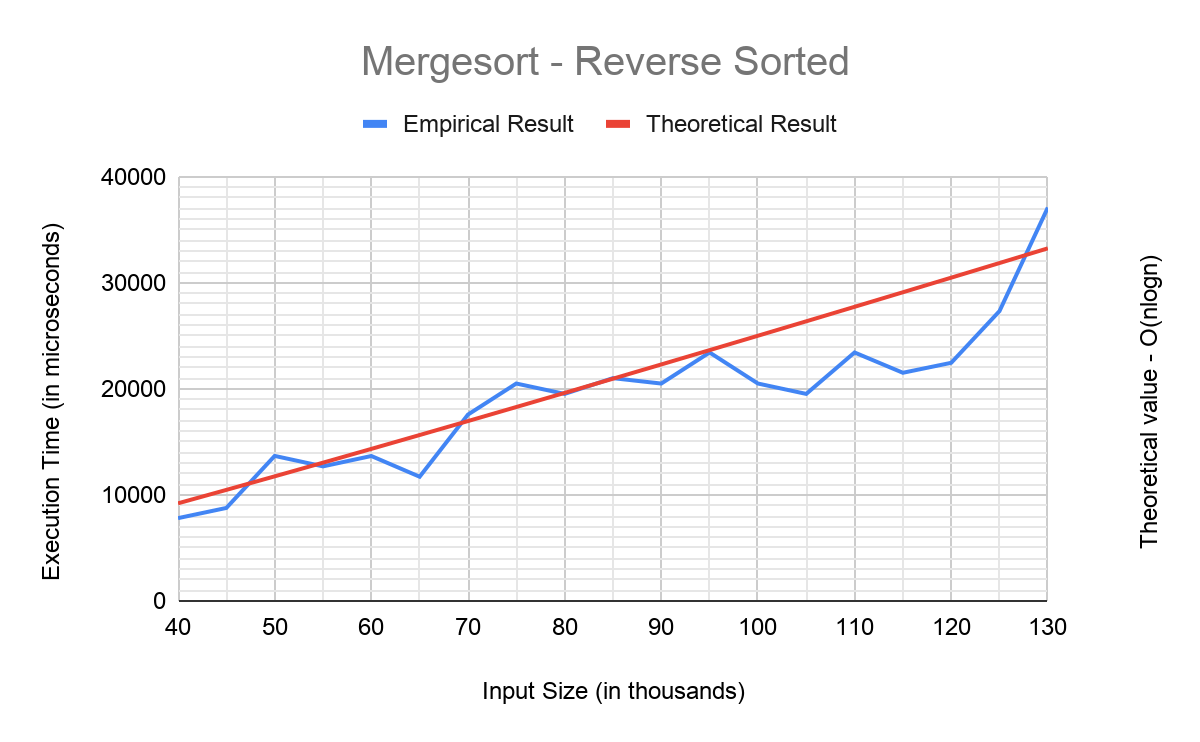
**1-Best-Case & Worst-Case:** Both worst and best case scenarios have the same time complexity which is O(nlog(n)). Actually this sorting algorithm has a tiny bit of difference between any case, so we can say that there aren’t any best or worst cases for Heapsort. Nevertheless we did experiments on sorted and reverse sorted arrays to show there is not much difference between each case. Here are the graphs for sorted and reverse sorted applied to heapsort.



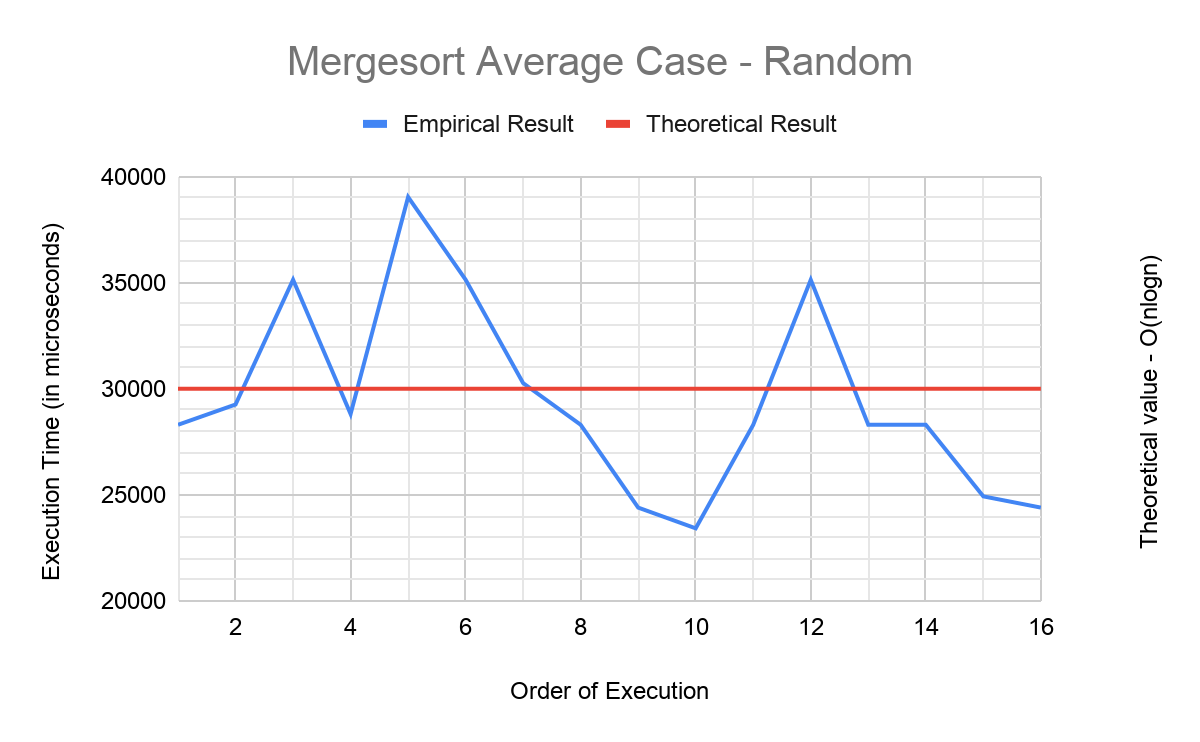
**2-Average-Case:** For the average case we make heapsort calculate randomly generated arrays with the same size (array size = 100000). Expected time complexity for the average case is O(nlog(n)). It looks like there are big differences between each execution but it’s time interval is so narrow that even small differences caused by randomness of the array makes it look big in this scale.

**MERGESORT**

**1-Best-Case & Worst-Case:** Both worst and best case scenarios have the same time complexity which is O(nlog(n)). It's pretty similar with Heapsort because there isn’t any difference between different cases in this algorithm too. Still here are the graphs for sorted and reverse sorted applied to heapsort.

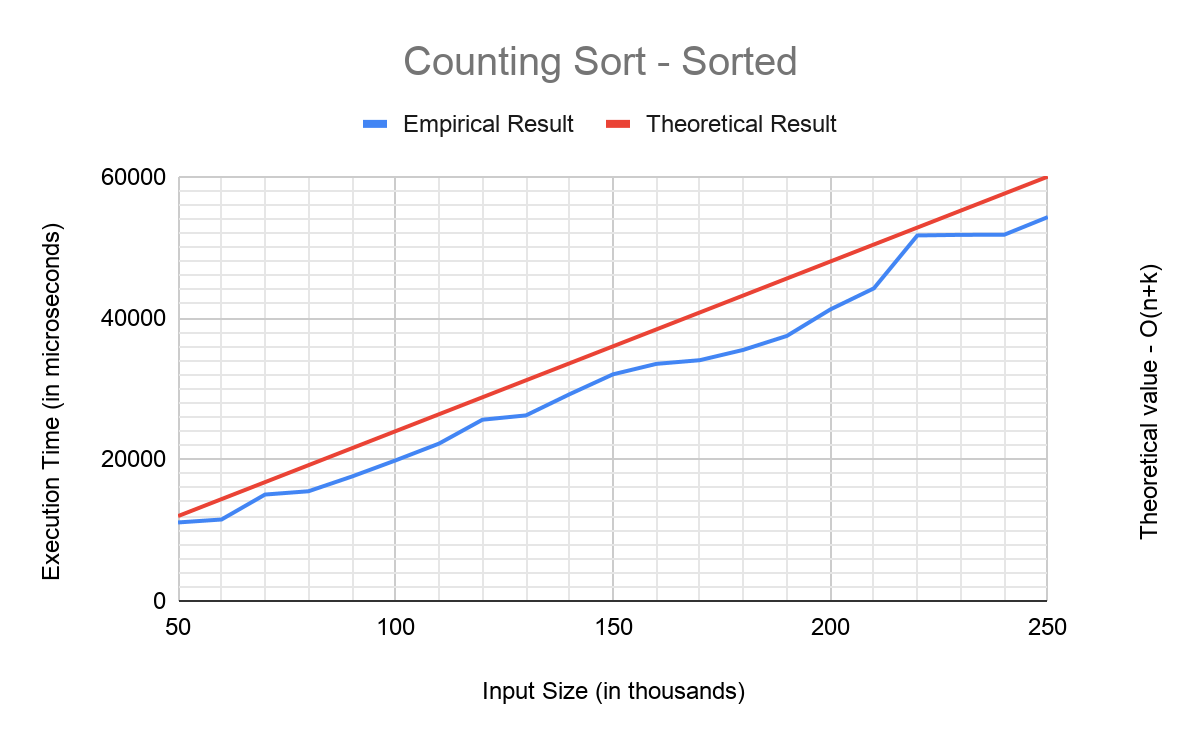
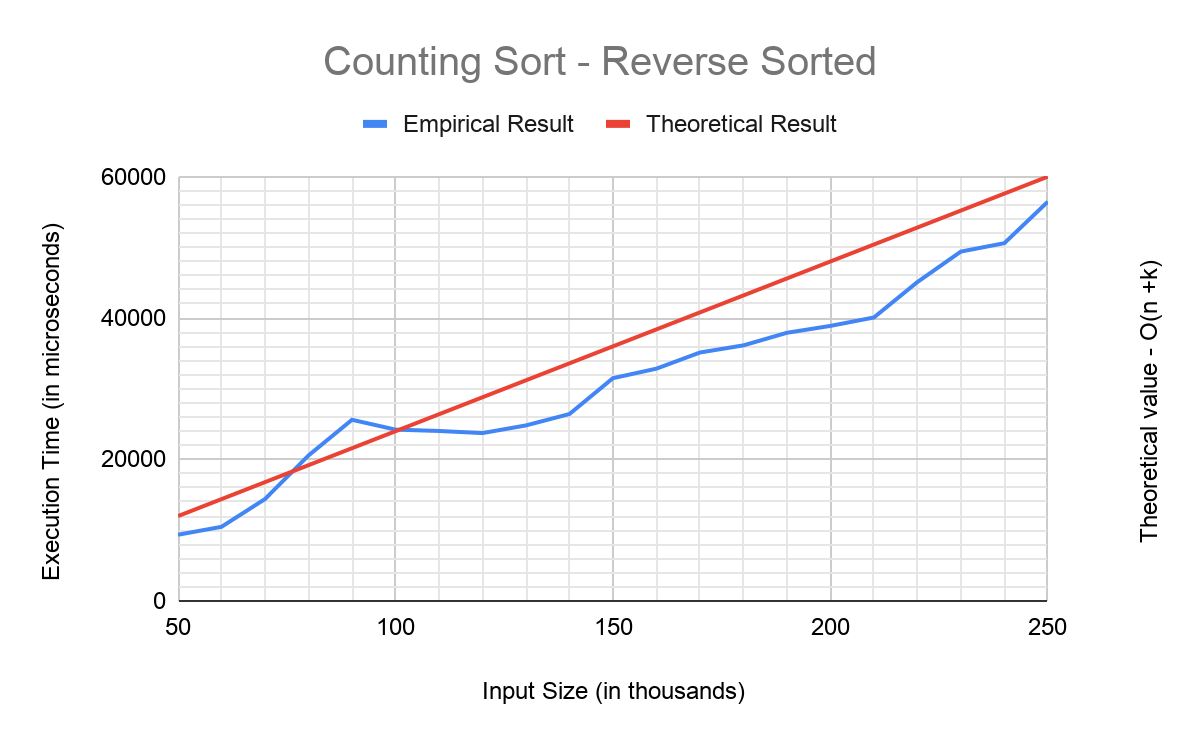


**2-Average-Case:** For the average case we make mergesort calculate randomly generated arrays with the same size (array size = 100000). Expected time complexity for the average case is O(nlog(n)). Again it’s the same with Heapsort in terms of different cases.

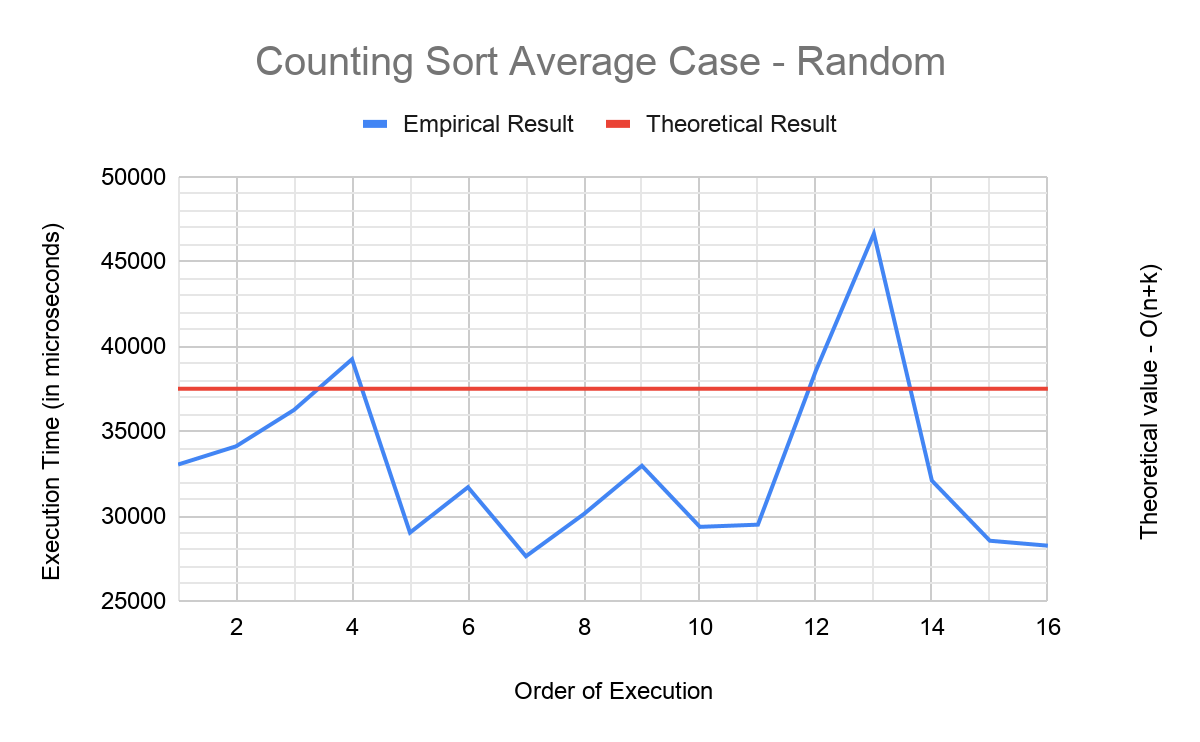


**COUNTING SORT**

**1-Best-Case & Worst-Case:** Since counting sort’s time complexity is O(n+k) and k is the range between maximum and minimum element of a given array, both best case and worst case have similar graphs.



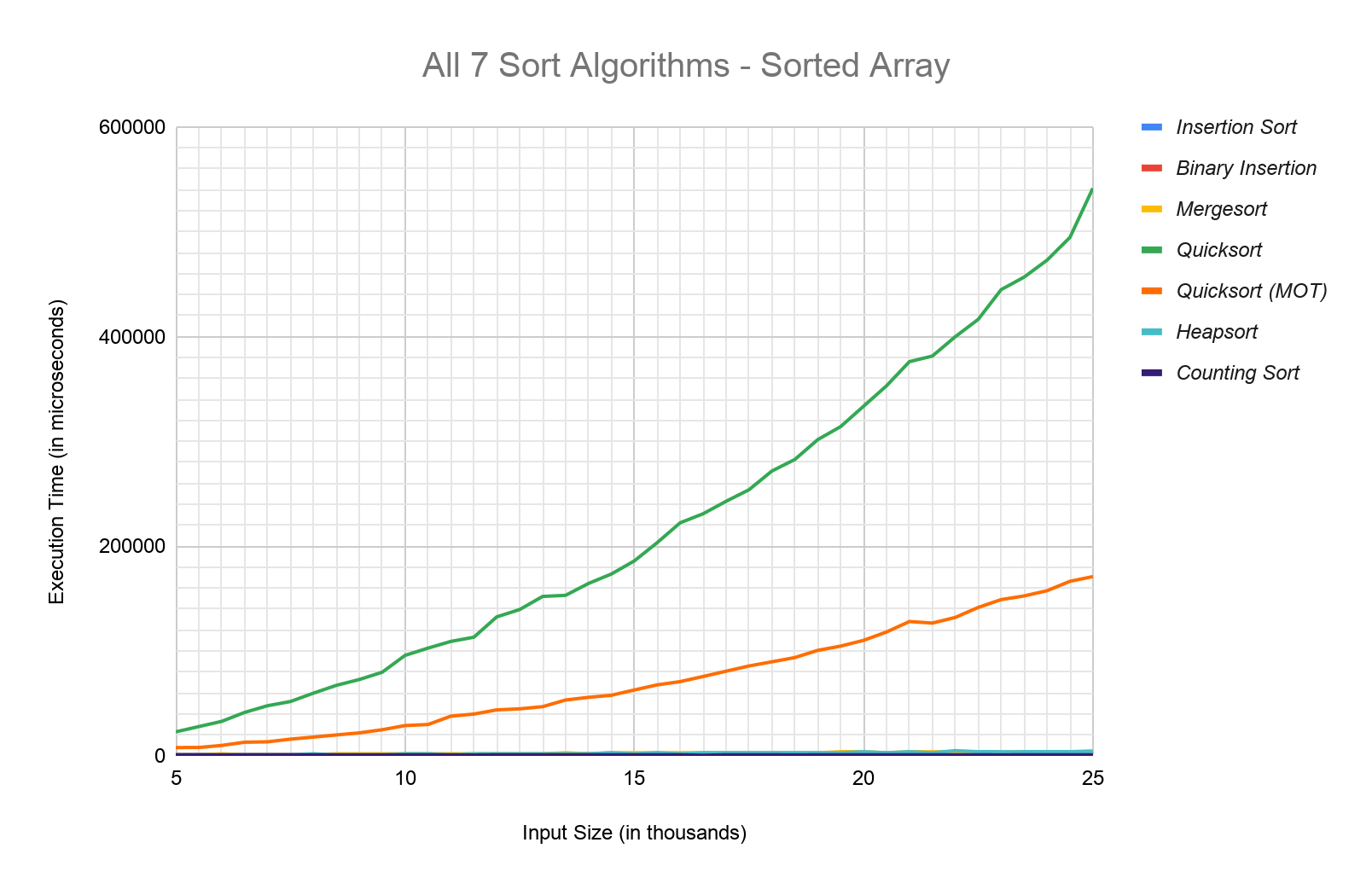
**2-Average-Case:** Since counting sort makes its time-memory trade-off on time side, it does not require anything that takes time that's why it's best, worst, and average case graphs are all similar. It’s time complexity is O(n+k) and k is the range between maximum and minimum element of a given array.

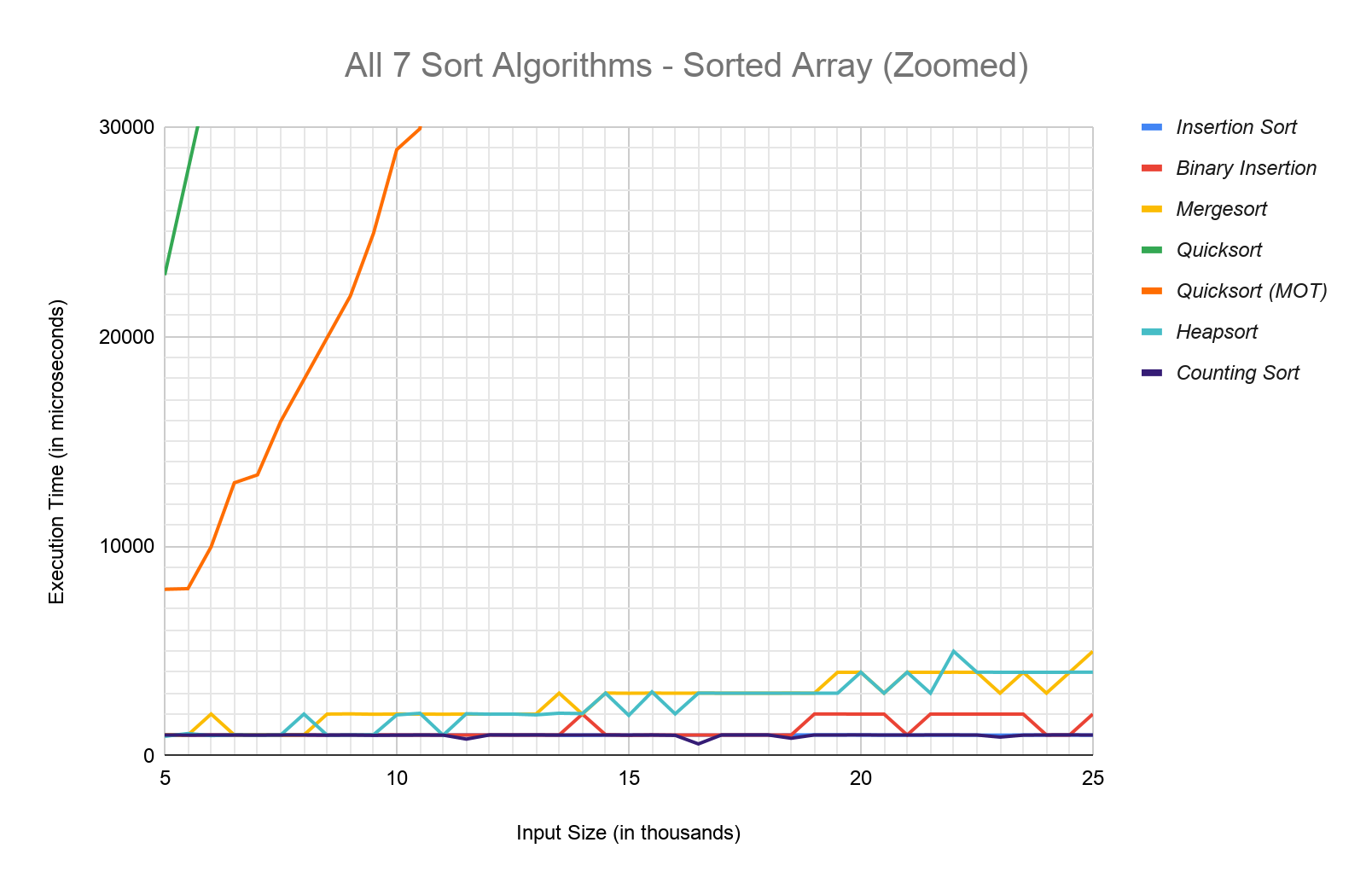


**CONCLUSION**

We also wanted to see every algorithms' performance in a single graph. In sorted arrays we expect that the quicksort algorithms to be the slowest ones and “median of three” is faster one in those two. Also we expect that counting sort and insertion sort to be the fastest ones among them. Actually they have very identical results but it’s because of the fact that they cost so little in terms of performance. Mergesort and Heapsort also have the same time complexity (O(nlogn)) and they have very similar graphs.

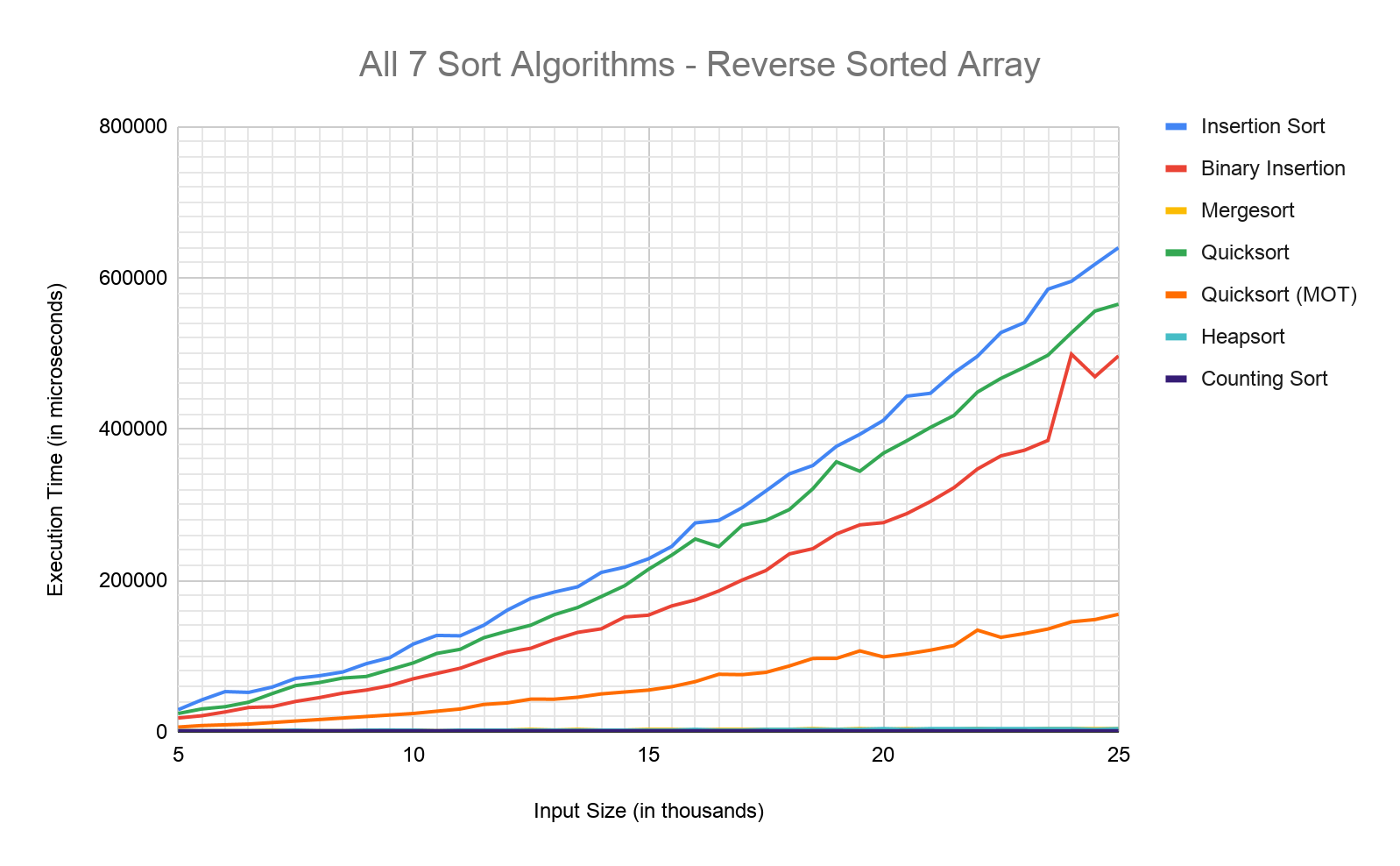
Comparing all given sorting algorithms for sorted arrays.



Zoomed-in version so we can see the better performing algorithms closer.

This time we expect the insertion sort to be the slowest and the quicksort algorithms and binary insertion sort again should be in that area. Also we should indicate that in this case all of them have O() time complexities. We can say that the worst case of the quicksort is still better than insertion sort but only the “median of three” quicksort is better than binary insertion sort. For the other ones counting sort is as expected the fastest one again. Again Mergesort and Heapsort also have the same time complexity (O(nlogn)) and again they have very similar graphs.

Comparison of all given algorithms on reverse-sorted arrays.



Zoomed-in version so we can see better performing algorithms closer.