MARMARA UNIVERSITY

FACULTY OF ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE ENGINEERING

CSE 2046 – ANALYSIS OF ALGORITHMS
COMPARING SORTING ALGORITHMS

STUDENTS	NUMBERS
Sueda BİLEN	150117044
Özge SALTAN	150517059
Zehra KURU	150119841

Submitted To:	Due Date:
Asst.Prof.Ömer Korçak	16/05/2021

Contents

Purpose of the Project

The main goal of this project is to design an experiment to compare the theoretical and empirical results of sorting algorithms. According to our observations, we will do some analysis with plots and tables.

Generating Inputs

1. Insertion Sort:

Insertion sort is a decrease and conquer algorithm and a simple sorting algorithm that is based on splitting an array into two parts as sorted and unsorted.

The best case is a sorted array in ascending order for insertion sort. The time complexity of the best case is O(n).

The worst case is a reversed sorted array in descending order for insertion sort. The time complexity of the worst case is $O(n^2)$.

For the average case, we considered it should be better than worst case and should be worse than better case, so it may be a random array for this sort. The time complexity of the average case is $O(n^2)$.

2. Binary Insertion Sort:

Binary insertion sort is used for reducing comparisons in the insertion sort algorithm. Therefore, this algorithm uses Insertion sort logic with binary search.

The time complexity of all cases is $O(n^2)$. Because the same logic is shared with the insertion sort, we used the same sample inputs with the insertion sort.

3. Merge Sort:

Merge sort is a divide and conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The time complexity of all cases is O(nlogn).

For the best case, we used a sorted array in ascending order algorithm to reduce the comparisons between the elements.

For the worst case, we found a permutation while researching the merge sort. This permutation parses the array into n parts and reorganizes according to the algorithm. We took inputs that have 2^n elements.

For the average case, we used random inputs that are generated with different sizes.

4. Quick Sort (pivot is always selected as the first element):

Quick sort is a divide and conquer algorithm. It selects the first element as the pivot and partitions the array.

For the best case, we generated inputs that include the median of the array as the first element. The time complexity of the best case is O(nlogn).

For the worst case, we generated inputs with reversed sorted and sorted arrays. Because the worst case occurs when the largest or smallest element is selected as the pivot. The time complexity of the worst case is $O(n^2)$.

For the average case, we used random inputs that are generated with different sizes. The time complexity of the average case is O(nlogn).

5. Quick Sort (with median-of-three pivot selection):

Quick sort is a divide and conquer algorithm. It selects the median of first, last and middle elements as pivot and partitions the array.

For the best case, we used a sorted array in ascending order input samples. The time complexity of the best case is O(nlogn).

For the worst case, we put the largest three elements of the array into the first, last, and middle positions of the input. The time complexity of the worst case is $O(n^2)$.

For the average case, we used random inputs that are generated with different sizes. The time complexity of the average case is O(nlogn).

6. Heap Sort:

Heap sort is a transform and conquer algorithm. Firstly this algorithm builds a max heap from the input array. Then, it removes the root element from the top and adds to the last index of the sorted array one by one.

The best case for the heap sort algorithm is the same element in all indexes of the input array. The time complexity of the best case is O(n) for this input case.

The worst case for the heap sort algorithm is the sorted array in ascending order. The time complexity of the worst case is O(nlogn) for this input case.

For the average case, we used random inputs that are generated with different sizes. The time complexity of the average case is O(nlogn).

7. Counting Sort:

Counting sort is a sorting technique based on keys between a specific range. It uses three arrays to compare and sort the element according to the range and key values.

Counting sort's time complexity is O(n+k). Therefore, both the range and input size are affecting the time. Because of this fact, we generated range-based and input size-based inputs with different values. It works best when the range of the input is not too wide.

Deciding on Metrics

There are two alternatives for deciding on efficiency metrics. First alternative is inserting counters into our program and counting the basic operations of sorting algorithms. Second alternative is to time our program for algorithms.

We decided to do the second alternative, we put the 'start time' before calling methods of algorithms and also 'end time' right after the methods. We used nano-seconds to measure the time difference because it is more precise than the millisecond. With this time scale, we observed the difference easily. To plot the measurements, we transformed nanoseconds to milliseconds for getting more understandable observations.

We repeated the execution of the experiment for each algorithm several times to get accurate results. After executions, we took the average of the results we obtained.

Analyzing Results

1. Insertion Sort:

For the best case sorted array in ascending order is appropriate to use. We observed sorted array inputs gave lowest values and time amount is about to increase n times when input size increased as expected. Our input sizes are 100, 1000, and 10000 (2 different inputs) for the best case.

Table 1 Insertion Sort Best Case

	n	ms_1	ms_2	ms_3	AVERAGE
best	100	0,0049	0,0057	0,0042	0,0049
best	1000	0,0222	0,0224	0,0224	0,0223
best	10000	0,2359	0,2052	0,1981	0,2131
best	10000	0,2083	0,2011	0,2073	0,2056

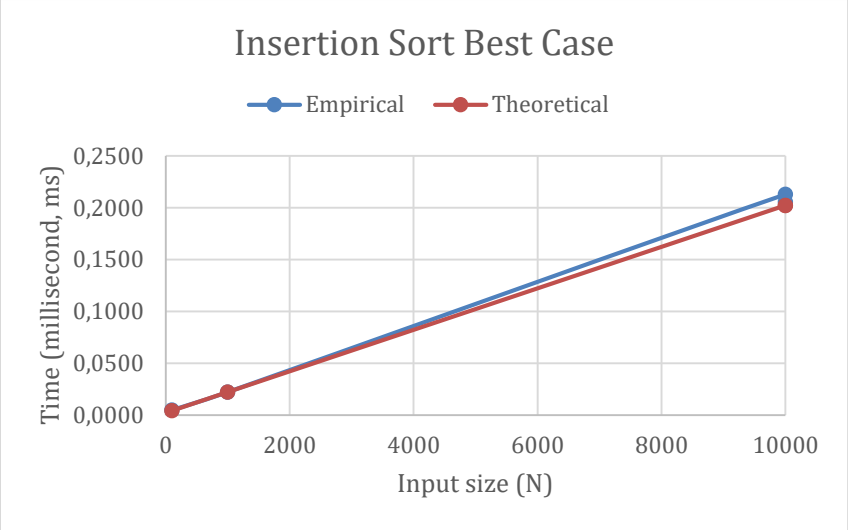

Figure 1 Insertion Best Case

For the average case, we used random inputs that we generated before. For this case, the time complexity is $O(n^2)$. The amount of the time measured is increased about to n^2 times when input size increased as expected. Our input sizes are 100, 500, 1000 (2 different inputs), and 10000 for the average case.

Table 2 Insertion Sort Average Case

	n	ms_1	ms_2	ms_3	AVERAGE
average	100	0,059	0,0846	0,0476	0,0637
average	500	0,8208	0,6355	0,6762	0,7108
average	1000	1,0186	0,9118	0,9967	0,9757
average	1000	0,9094	1,4694	1,0056	1,1281
average	10000	42,2698	43,5691	40,853	42,2306

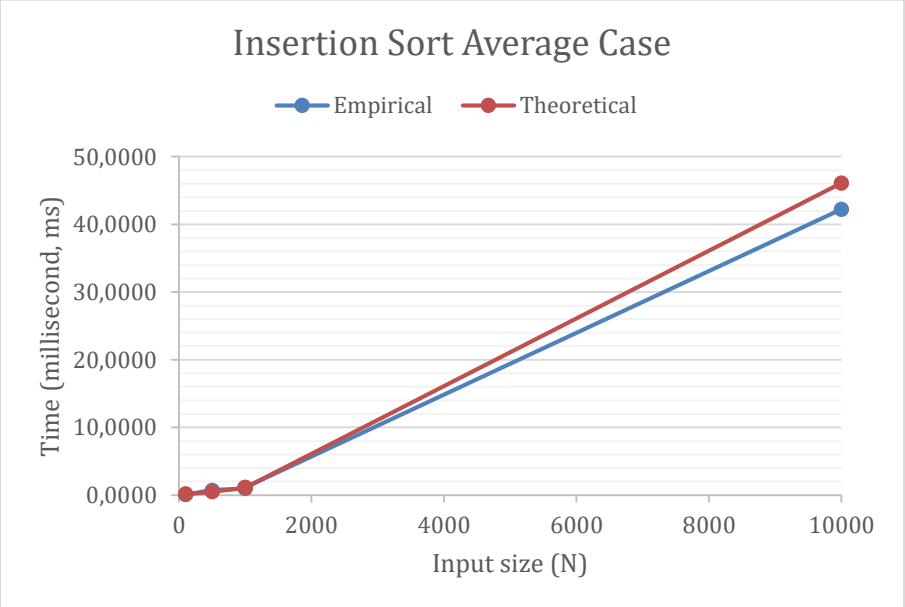

Figure 2 Insertion Sort Average Case

For the worst case, when the array is reserved sorted, in the other words when the array is sorted as a descending order, it happens worst case in insertion sort. As it is seen in the table, we used input sizes for 100 (two different input samples), 1000, and 10000, and the time was increased as 0.11, 0.09, 1.188, and 83.60. It increases a lot more as input sizes are increased as expected. Our observations match with the theory (i.e. quadratic).

Table 3 Insertion Sort Worst Case

	n	ms_1	ms_2	ms_3	AVERAGE
worst	100	0,0851	0,1475	0,1068	0,1131
worst	100	0,0824	0,1088	0,0823	0,0912
worst	1000	2,2221	1,6747	1,7494	1,8821
worst	10000	87,5626	81,3531	81,8951	83,6036

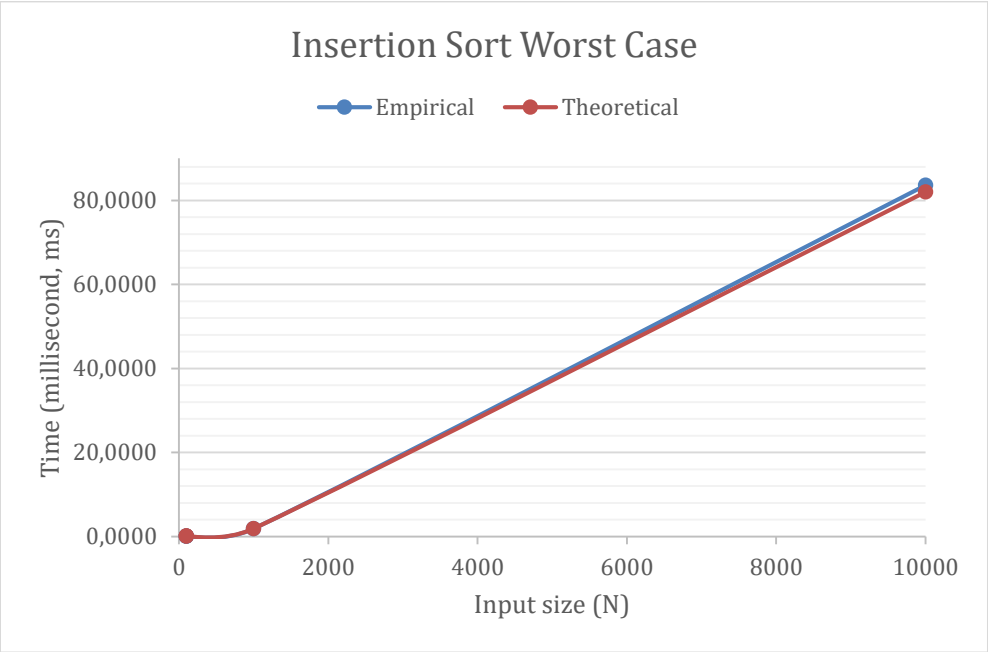

Figure 3 Insertion Sort Worst Case

.

2. Binary Insertion Sort:

For the best case, a sorted array in ascending order is appropriate to use. We observed the sorted array gives the lowest values and the time amount is about to increase n times when input size increased as expected. When we compare this algorithm with insertion sort with time perspective there is just a minor difference, however binary insertion sort is quicker than insertion sort.

Table 4 Binary Insertion Sort Best Case

	n	ms_1	ms_2	ms_3	AVERAGE
best	100	0,0038	0,0035	0,0038	0,0037
best	1000	0,0146	0,0148	0,015	0,0148
best	10000	0,1439	0,1339	0,1589	0,1456
best	10000	0,1345	0,1321	0,1288	0,1318

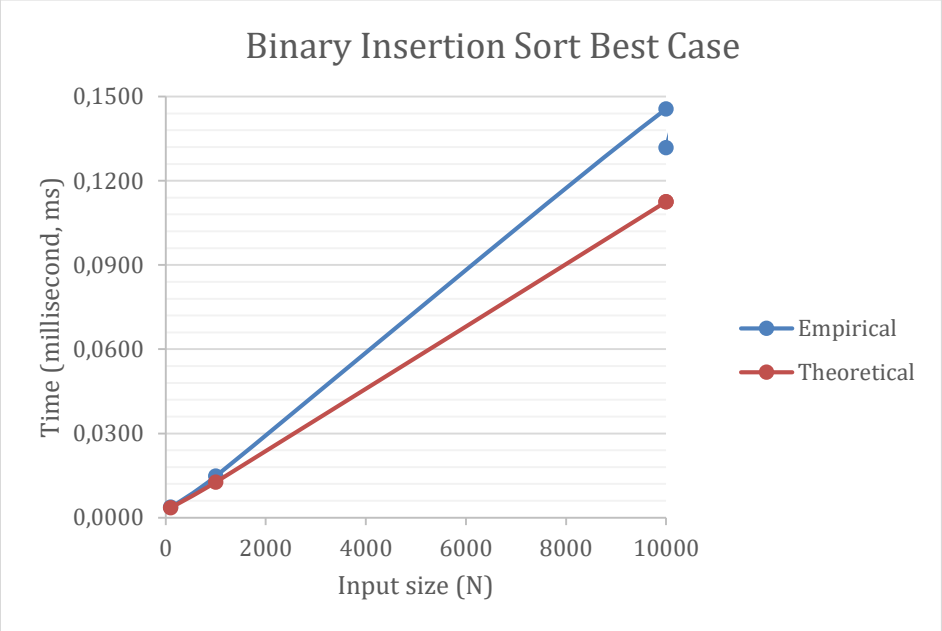

Figure 4 Binary Insertion Sort Best Case

For the average case, we used random inputs that we generated before. For this case, the time complexity is $O(n^2)$. The amount of the time measured is increased about to n^2 times when input size increased as expected. Our input sizes are 100, 500, 1000 (2 different inputs), and 10000 for the average case.

Table 5 Binary Insertion Average Case

	n	ms_1	ms_2	ms_3	AVERAGE
average	100	0,1636	0,1909	0,1379	0,1641
average	500	0,9193	1,4695	1,4196	1,2695
average	1000	2,0251	3,3356	1,7677	2,3761
average	1000	1,8074	2,4025	1,5993	1,9364
average	10000	142,3138	124,2853	125,8336	130,8109

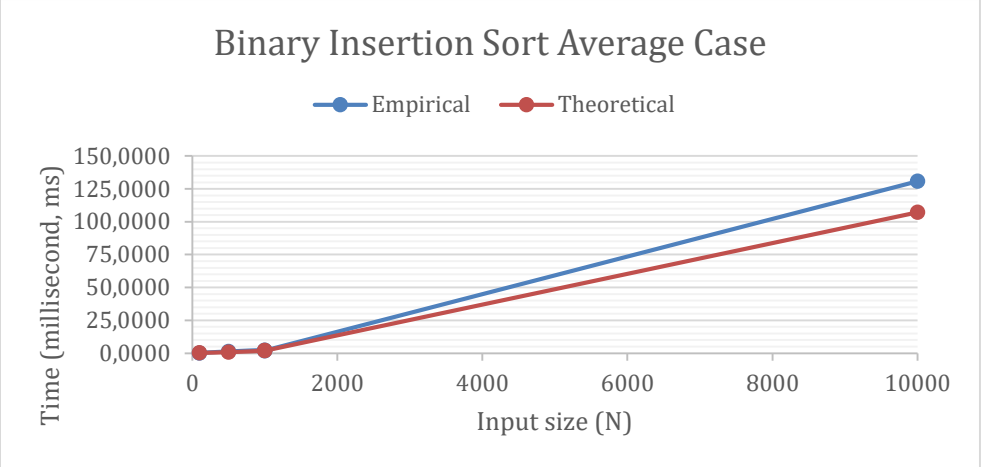

Figure 5 Binary Insertion Average Case

Since binary insertion sort shares the same logic with insertion sort, we use the same sample inputs for the worst-case too, so the array is reversed order. As it is shown in our table, we used input sizes for 100 (two different input samples), 1000, and 10000, and the time was increased as 0.15, 0.17, 2.15, and 114.19. It increases as quadratic. Even though binary insertion sort is better than insertion sort for best case, it is the same for the worst case in time comparisons.

Table 6 Binary Insertion Sort Worst Case

	n	ms_1	ms_2	ms_3	AVERAGE
worst	100	0,1394	0,1756	0,1756	0,1511
worst	100	0,1674	0,2153	0,1406	0,1744
worst	1000	1,9935	2,1708	2,3145	2,1596
worst	10000	126,9803	107,1052	108,4793	114,1883

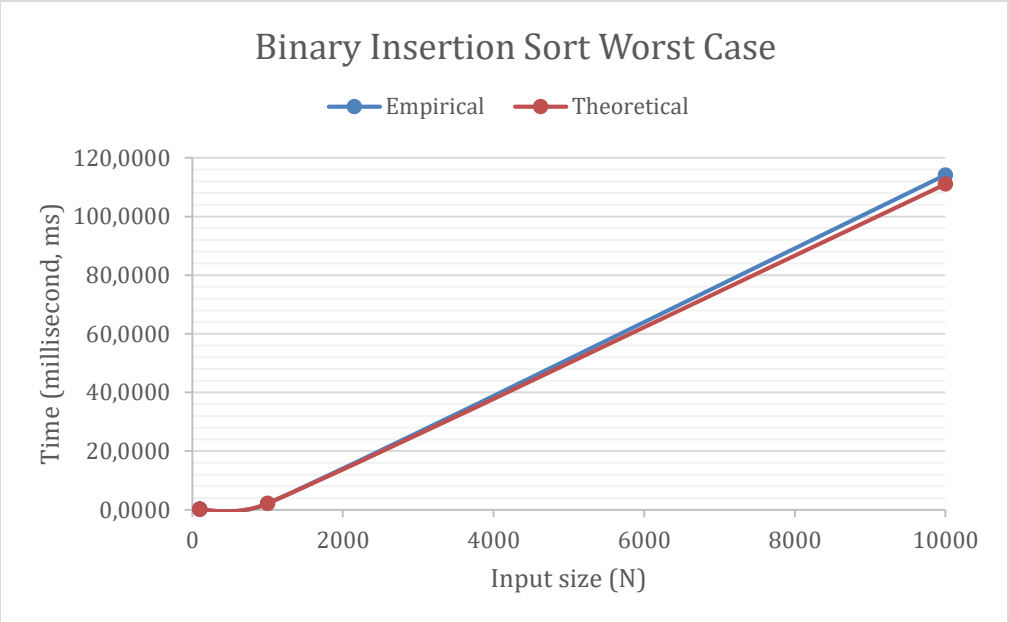

Figure 6 Binary Insertion Sort Worst Case

3. Merge Sort:

The time complexity for merge sort is O(nlogn) in all cases. To get the best time, a sorted array in ascending order is appropriate to use in this case. We observed the sorted array gives the lowest values and the time amount is about to increase nlogn times when the input size is increased as expected.

Table 7 Merge Sort Best Case

	n	ms_1	ms_2	ms_3	AVERAGE
Best	100	0,0494	0,0647	0,0483	0,0541
Best	1000	0,4932	0,5116	0,6053	0,5367
Best	10000	1,0289	1,4108	1,429	1,2896
Best	10000	1,5671	1,1523	1,1403	1,2866

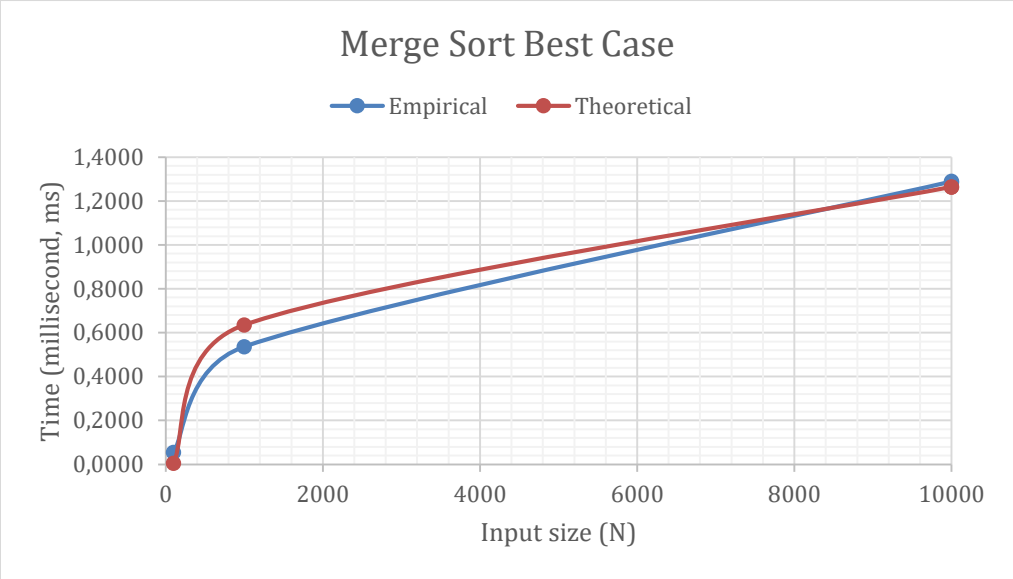

Figure 7 Merge Sort Best Case

For the average case, we used random inputs that we generated before. For this case, the time complexity is O(nlogn). The amount of the time measured is increased about to nlogn times when input size increased as expected. Our input sizes are 100, 500, 1000 (2 different inputs), and 10000 for the average case.

Table 8 Merge Sort Average Sort

	n	ms_1	ms_2	ms_3	AVERAGE
average	100	0,0526	0,0595	0,0541	0,0554
average	500	0,3355	0,3687	0,3964	0,3669
average	1000	1,0821	1,2169	0,8873	1,0621
average	1000	0,6607	0,7083	1,0141	0,7944
average	10000	2,7663	1,8212	2,9753	2,5209

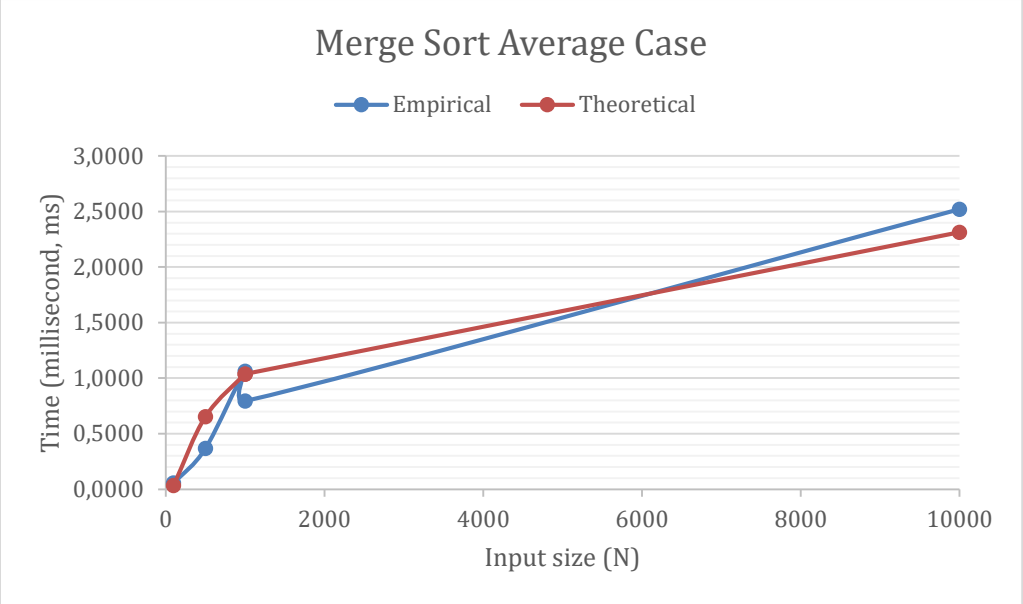

Figure 8 Merge Sort Average Case

For the worst case, we used inputs that we generated according to the permutations we found in researching. We observed the highest time measurement in this case as expected. The results are showing an algorithm about to increase nlogn times while input size is enlarging.

Table 9 Merge Sort Worst Case

	n	ms_1	ms_2	ms_3	AVERAGE
worst	128	0,1091	0,0647	0,3164	0,1634
worst	1024	0,5817	0,4941	0,524	0,5333
worst	10000	3,4395	3,0009	3,2236	3,2213

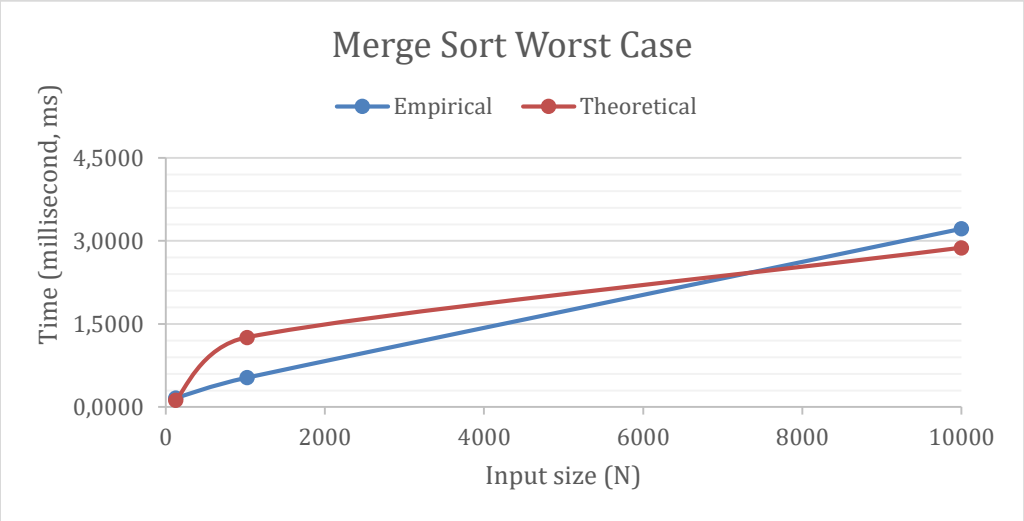

Figure 9 Merge Sort Worst Case

4. Quick Sort (pivot is always selected as the first element):

For the best case, we tried inputs with the median of the array as the leftmost element. These inputs gave us the lowest time measurement in this case as expected, and results are showing that the algorithm is about to increase nlogn times while input size is enlarging.

Table 10 Quick Sort Best Case (first-pivot)

	n	ms_1	ms_2	ms_3	AVERAGE
best	128	0,0572	0,059	0,0374	0,0512
best	1024	0,4886	0,5772	0,7644	0,6101
best	8192	5,1953	3,9172	6,2669	5,1265

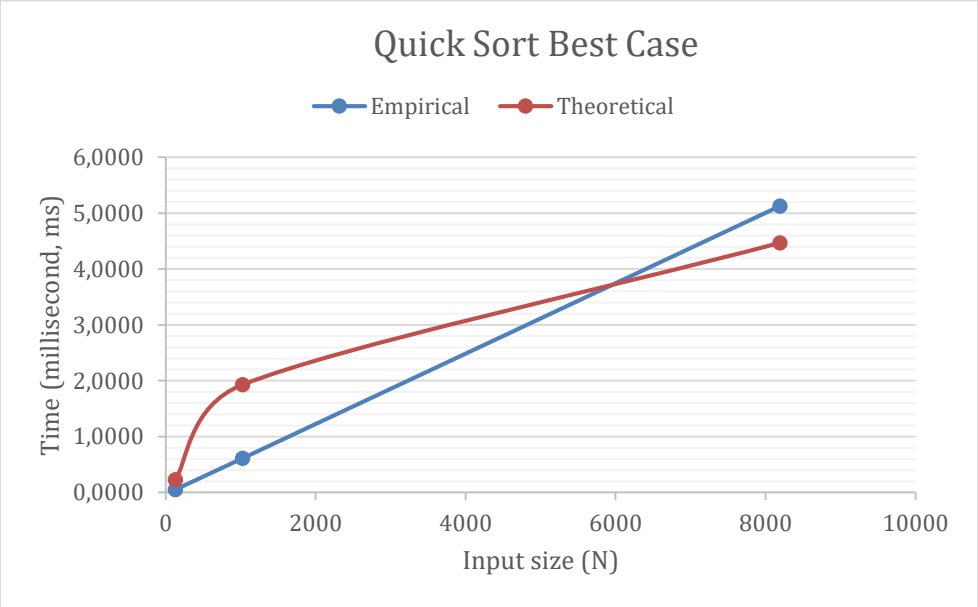

Figure 10 Quick Sort Best Case

For the average case, we used random inputs that we generated before. For this case, the time complexity is $O(n^2)$. The amount of the time measured is increased about to n^2 times when input size increased as expected. Our input sizes are 100, 500, 1000 (2 different inputs), and 10000 for the average case.

Table 11 Quick Sort Average Case (first-pivot)

	n	ms_1	ms_2	ms_3	AVERAGE
average	100	0,4938	0,5239	0,5306	0,5161
average	500	0,5431	0,6536	0,725	0,6406
average	1000	1,0033	1,3979	1,3728	1,2580
average	1000	1,1921	1,4281	1,5107	1,3770
average	10000	4,3861	6,299	7,524	6,0697

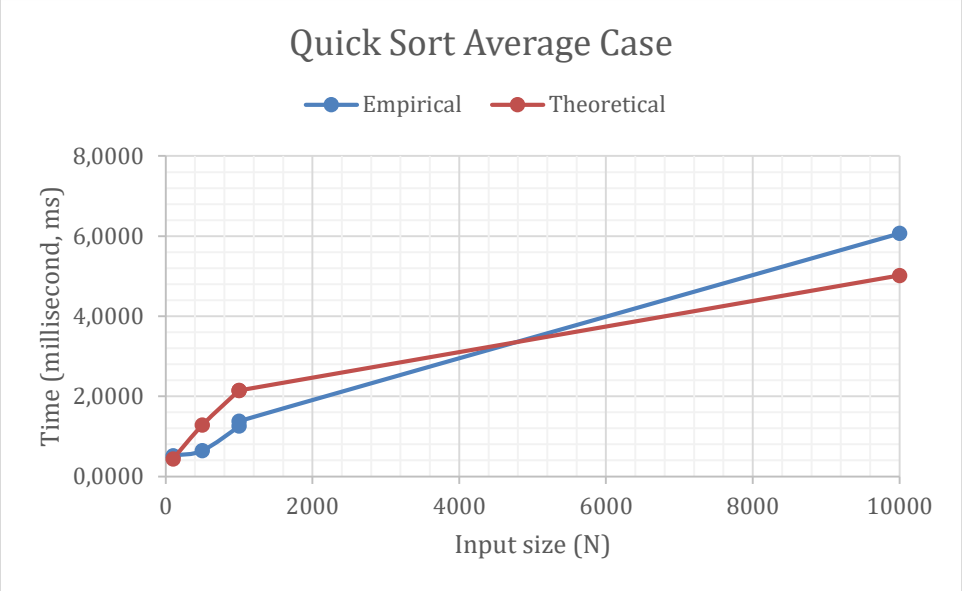

Figure 11 Quick Sort Average Case

For the worst case, we used inputs with the smallest element as the leftmost position in the input array. We observed the highest time measurement in this case as expected. The results are showing that the algorithm is about to increase n^2 times while input size is enlarging. Results coincide with time complexity $O(n^2)$.

Table 12 Quick Sort Worst Case (first-pivot)

	n	ms_1	ms_2	ms_3	AVERAGE
worst	100	7,5077	7,2613	7,3512	7,3734
worst	100	8,9706	8,5996	7,5075	8,3592
worst	1000	8,795	9,2219	9,6294	9,2154
worst	10000	10,1769	11,6078	15,9468	12,5772

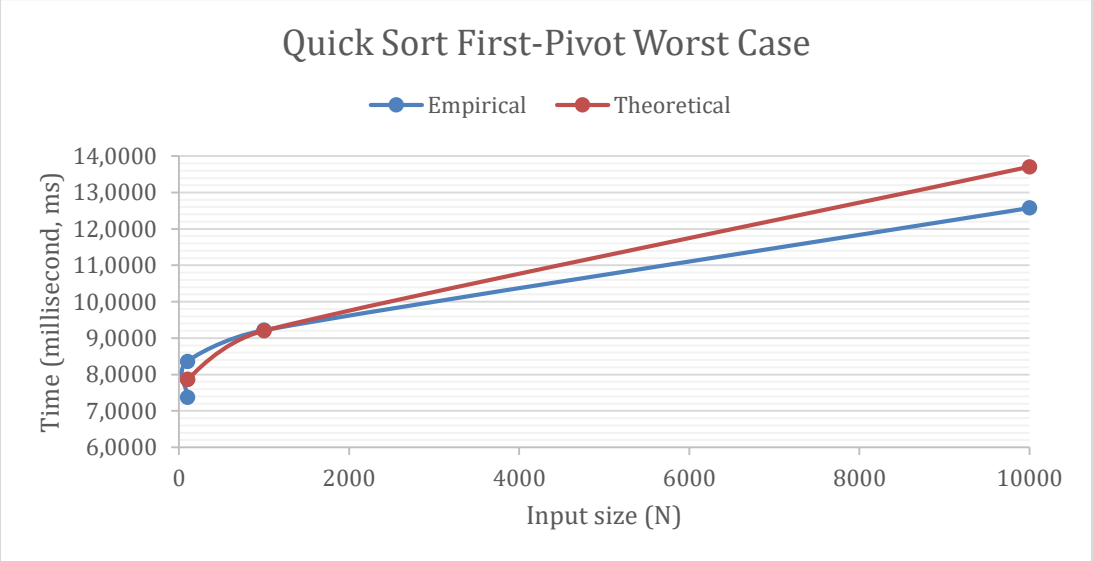

Figure 12 Quick Sort First-Pivot Worst Case

5. Quick Sort (with median-of-three pivot selection):

In median-of-three pivot quicksort, the best case happens when the median element is between three elements. Therefore we used the same input list with quicksort first pivots input list. The time complexity for the best case is O(nlogn). These inputs gave us the lowest time measurement in this case as expected, and results are showing that the algorithm is about to increase nlogn times while input size is enlarging. However, we observed quick sort algorithms work better on small arrays for all cases.

Table 13 Quick Sort Best Case (median-of-three pivot)

	n	ms_1	ms_2	ms_3	AVERAGE
best	100	0,0257	0,0476	0,0629	0,0454
best	1000	1,1293	1,2556	1,1262	1,1704
best	10000	21,9619	21,6788	29,657	24,4326
best	10000	27,9535	28,5874	26,6593	27,7334

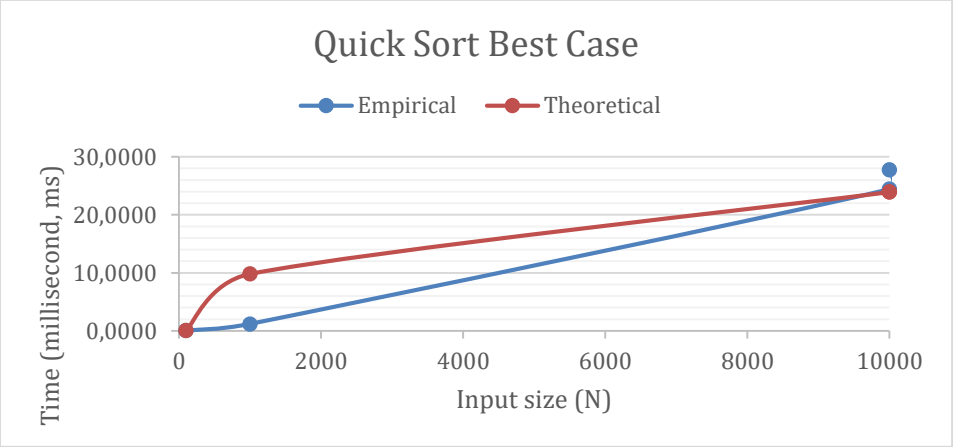

Figure 13 12 Quick Sort Best Case

For the average case, we used random inputs that we generated before. For this case, the time complexity is O(nlogn). The amount of the time measured is increased about to nlogn times when input size increased as expected. Our input sizes are 100, 500, 1000 (2 different inputs), and 10000 for the average case.

Table 14 Quick Sort Average Case (median-of-three pivot)

	n	ms_1	ms_2	ms_3	AVERAGE
average	100	0,0369	0,0487	0,0405	0,0420
average	500	0,2385	0,1437	0,1866	0,1896
average	1000	1,1517	0,8106	1,0982	1,0202
average	1000	0,7192	0,9459	0,621	0,7620
average	10000	3,0043	3,9645	3,0293	3,3327

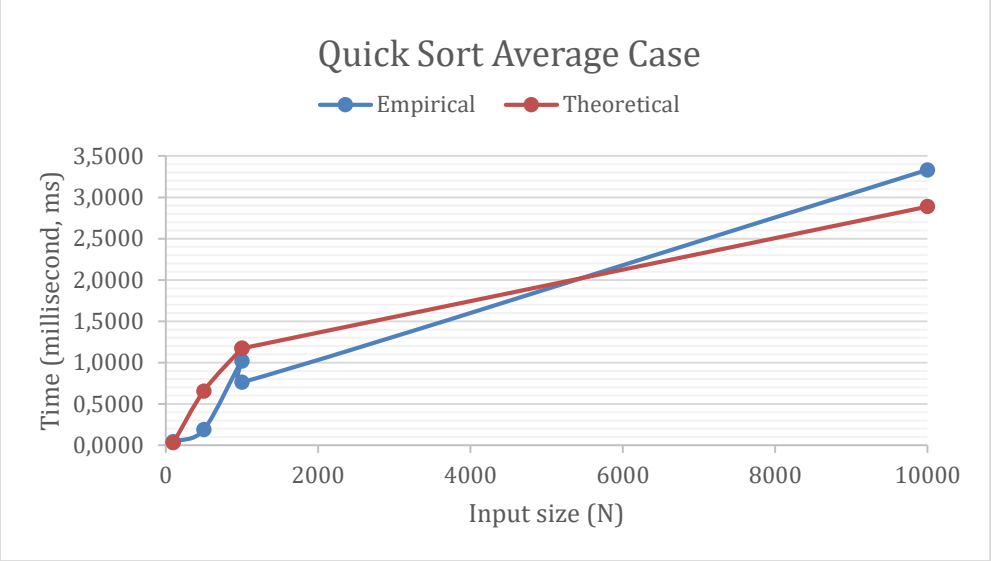

Figure 14 Quick Sort Average Case

For the worst case, we used inputs with the largest three elements in the first, middle, and last positions in the input array. We observed the highest time measurement in this case as expected. The results are showing that the algorithm is about to increase n^2 times while input size is enlarging. Results coincide with time complexity $O(n^2)$.

Table 15 Quick Sort Worst Case (median-of-three pivot)

	n	ms_1	ms_2	ms_3	AVERAGE
worst	100	0,282	0,248	0,2888	0,2729
worst	1000	4,0992	4,4394	5,3419	4,6268
worst	9999	30,0628	40,5206	39,7132	36,7655

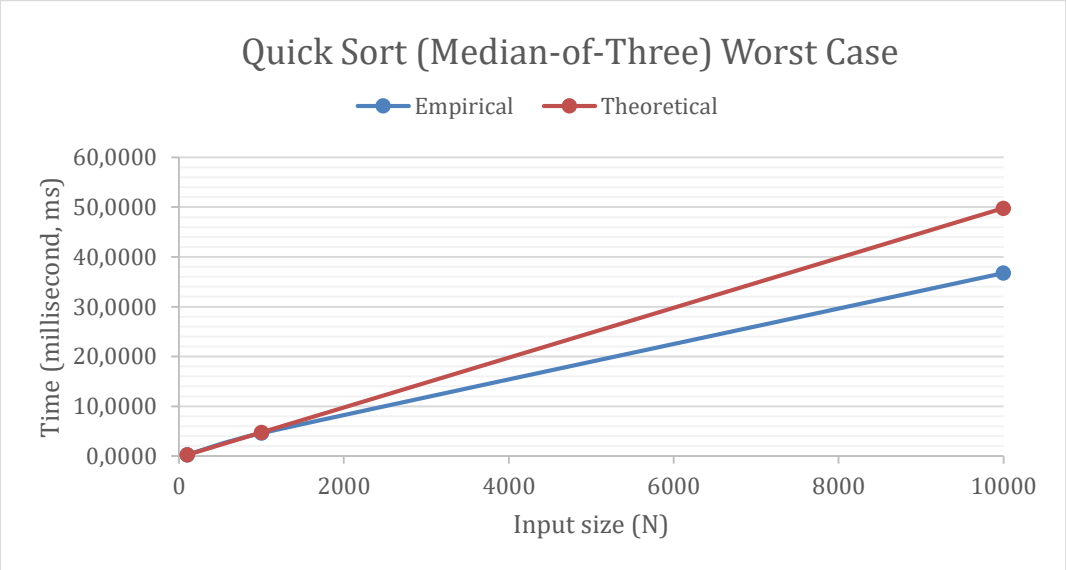

Figure 15 Quick Sort (Median-of-Three) Worst Case

6. Heap Sort:

For the best case, it takes less time when the elements are the same values since there are just comparisons and not swapping. It increases linearly as input sizes are increased in our table (i.e. O(n)). In our results, when we look at input samples as equal keys, random keys, and sorted array, we obtained that equal keys are the best case for heap sort.

Table 16 Heap Sort Best Case

	n	ms_1	ms_2	ms_3	AVERAGE
best	100	0,0142	0,0173	0,0323	0,0213
best	1000	0,0768	0,0809	0,0766	0,0781
best	10000	1,1363	0,775	1,0367	0,9827

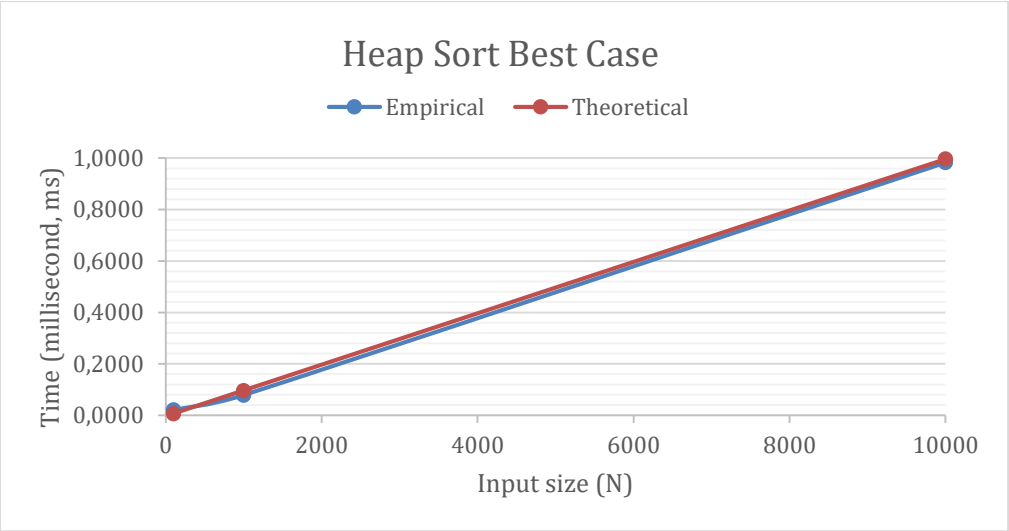

Figure 16 Heap Sort Best Case

For the average case, we used random input samples that we generated before. Since inputs are random keys, it has comparisons and swapping for max heap so it has more time to finish sorting. As it is shown in the table, it increases logarithmically as input sizes are increased. Our input sizes are 100, 500, 1000 (2 different inputs), and 10000 for the average case.

Table 17 Heap Sort Average Case

	n	ms_1	ms_2	ms_3	AVERAGE
average	100	0,046	0,0401	0,0536	0,0466
average	500	0,2707	0,4263	0,2761	0,3244
average	1000	0,5139	0,6003	0,3854	0,4999
average	1000	0,5571	0,5525	0,4928	0,5341
average	10000	2,9227	2,3476	1,9031	2,3911

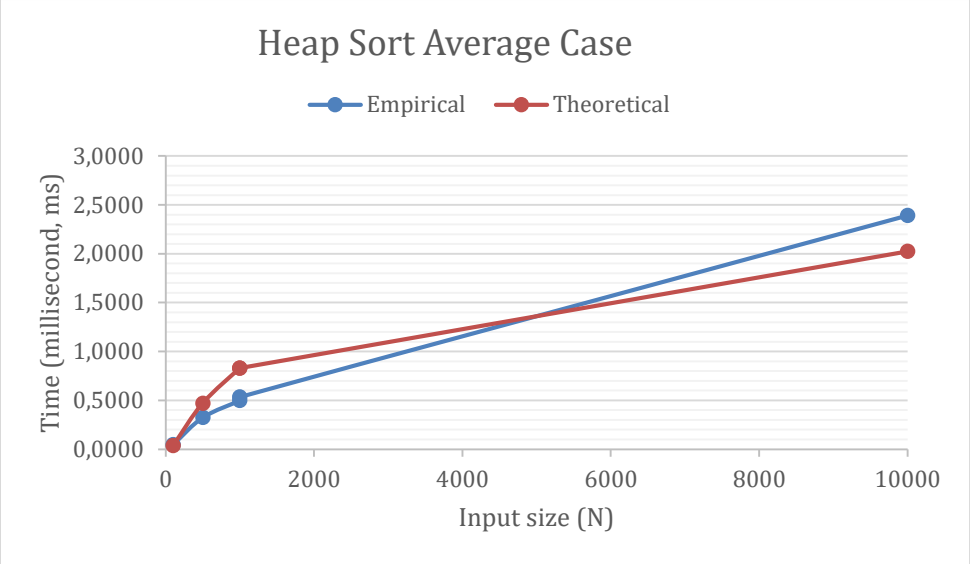

Figure 17 Heap Sort Average Case

For the worst case, we used a sorted array in ascending order since it uses max heap and it has to swap the key one by one in every insertion. When we look at the table of our results, it increases logarithmically as input sizes are increased. Our results match with the theoretical values (i.e. nlogn). Even though a sorted array is the worst case for this sorting algorithm, the results of a sorted array (worst case input sample) and random keys (average case input sample) are approximate to each other.

Table 18 Heap Sort Worst Case

	n	ms_1	ms_2	ms_3	AVERAGE
worst	100	0,0785	0,048	0,0634	0,0633
worst	1000	0,5784	0,4585	0,7103	0,5824
worst	10000	1,83	1,4119	1,801	1,6810
worst	10000	1,9041	1,6865	1,7695	1,7867

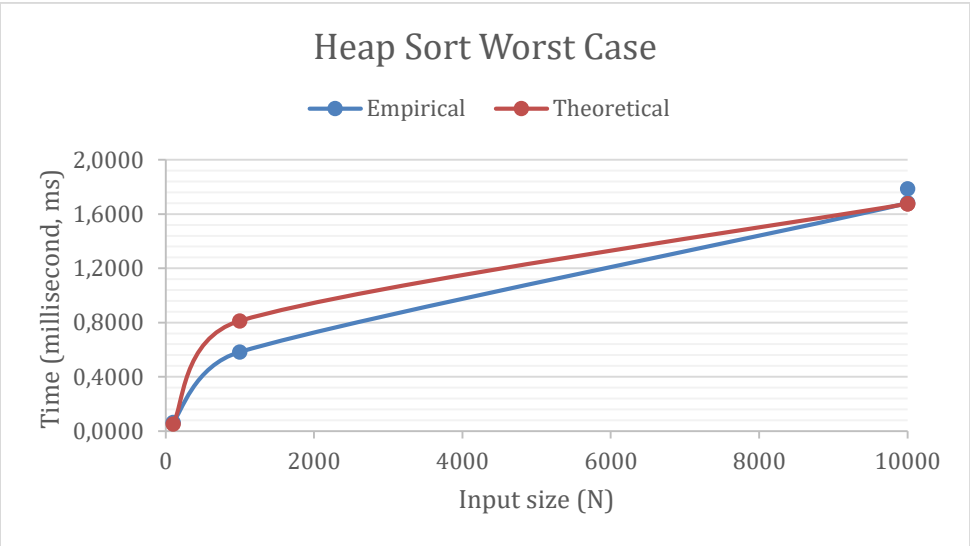

Figure 138 Heap Sort Worst Case

7. Counting Sort:

Counting sort is an algorithm different from other sorting algorithms. Because the time complexity value is fixed to 2 variables. These are input size and range values.

First, we observed how the time changed by keeping the input constant for you, changing the range values. As the range values increased, an increase was observed in the time values.

Table 19 Counting Sort fixed input size

n	k	ms_1	ms_2	ms_3	AVERAGE
100	200	0,0184	0,0266	0,0118	0,0189
100	500	0,0198	0,0336	0,0361	0,0298
100	1000	0,0281	0,0357	0,0425	0,0354
100	2000	0,1675	0,127	0,1509	0,1485
100	5000	0,305	0,2962	0,1127	0,2380
100	10000	0,2657	0,414	0,2284	0,3027

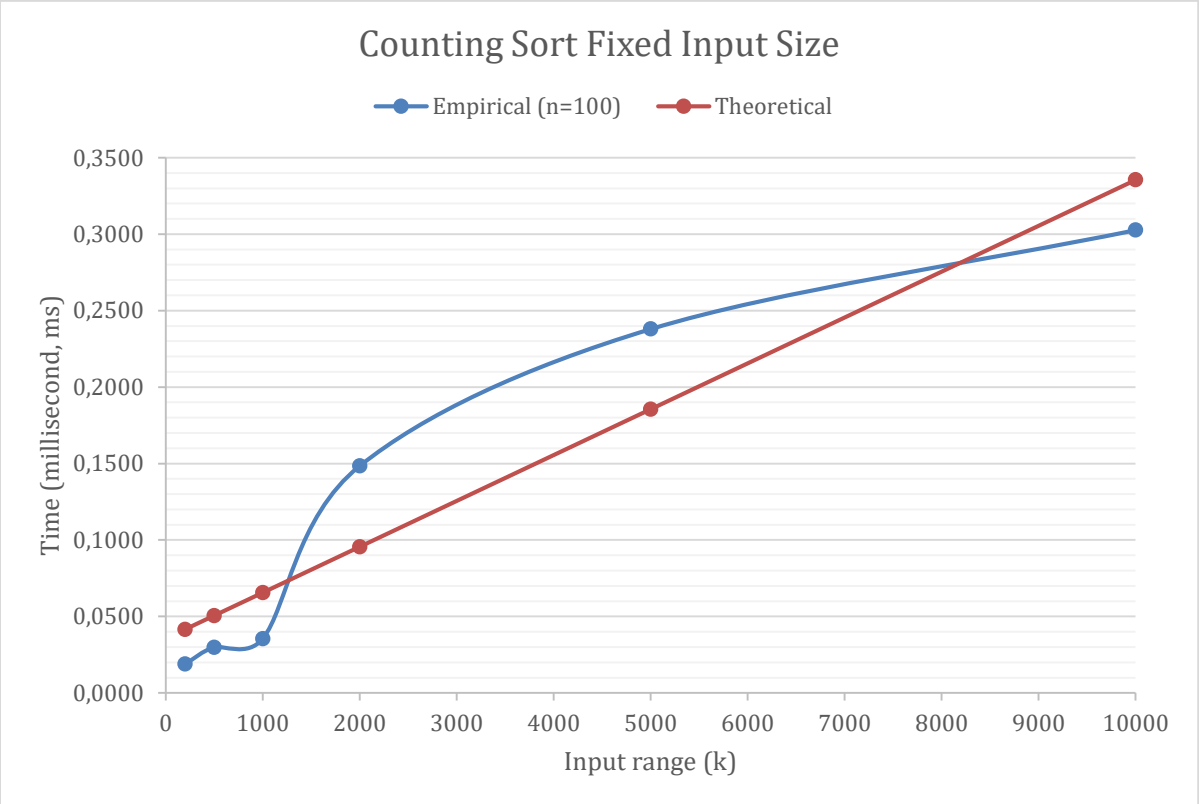

Figure 19 Counting Sort Fixed Input Size

Second, it was observed how the time changed by keeping the range value constant and changing the input size values. As the input values increased, an increase was observed in the time values.

Table 20 Counting Sort fixed range

n	k	ms_1	ms_2	ms_3	AVERAGE
100	10000	0,1268	0,1198	0,1197	0,1221
500	10000	0,1427	0,1465	0,143	0,1441
1000	10000	0,1707	0,1633	0,1723	0,1688
5000	10000	0,3495	0,355	0,3233	0,3426
10000	10000	0,5636	0,5117	0,6024	0,5592
10000	10000	0,5031	0,3163	0,3495	0,3896

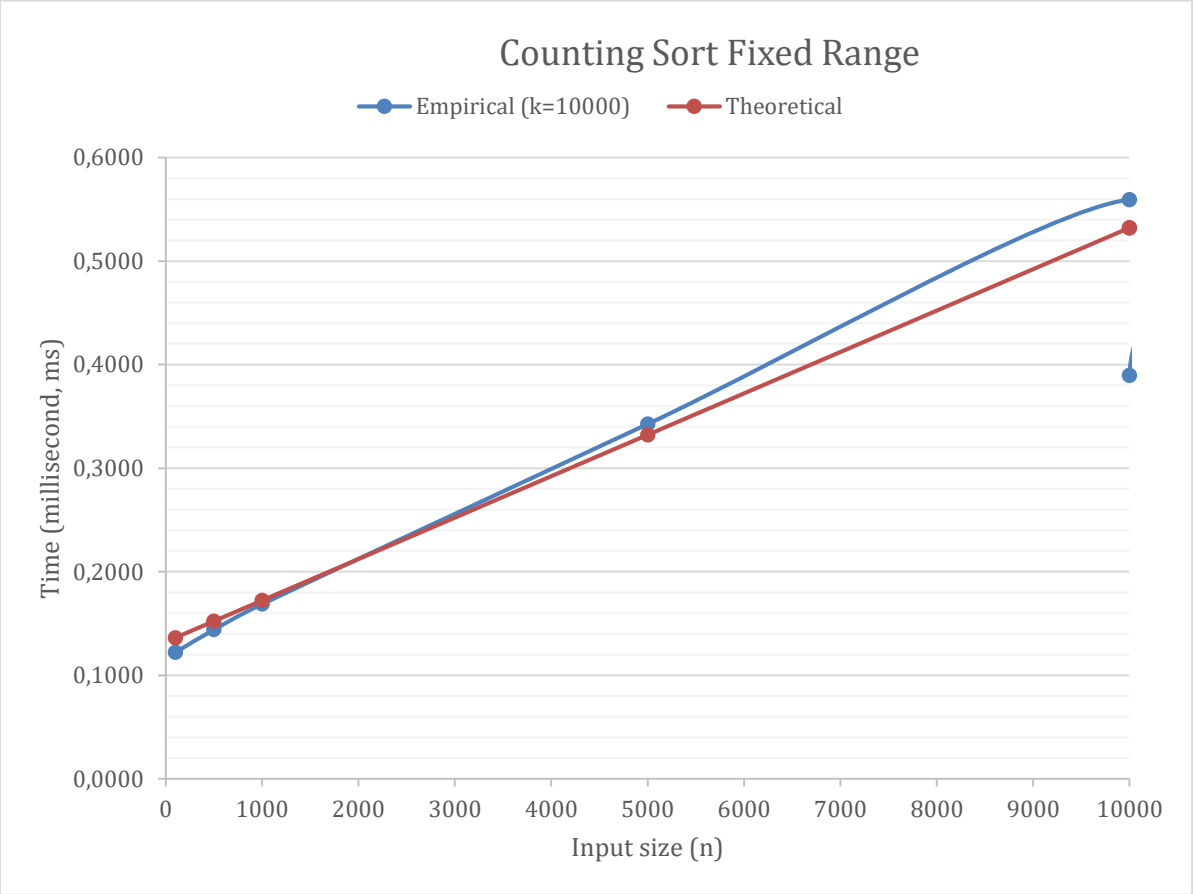

Figure 20 Counting Sort Fixed Range

Comparing All Algorithms

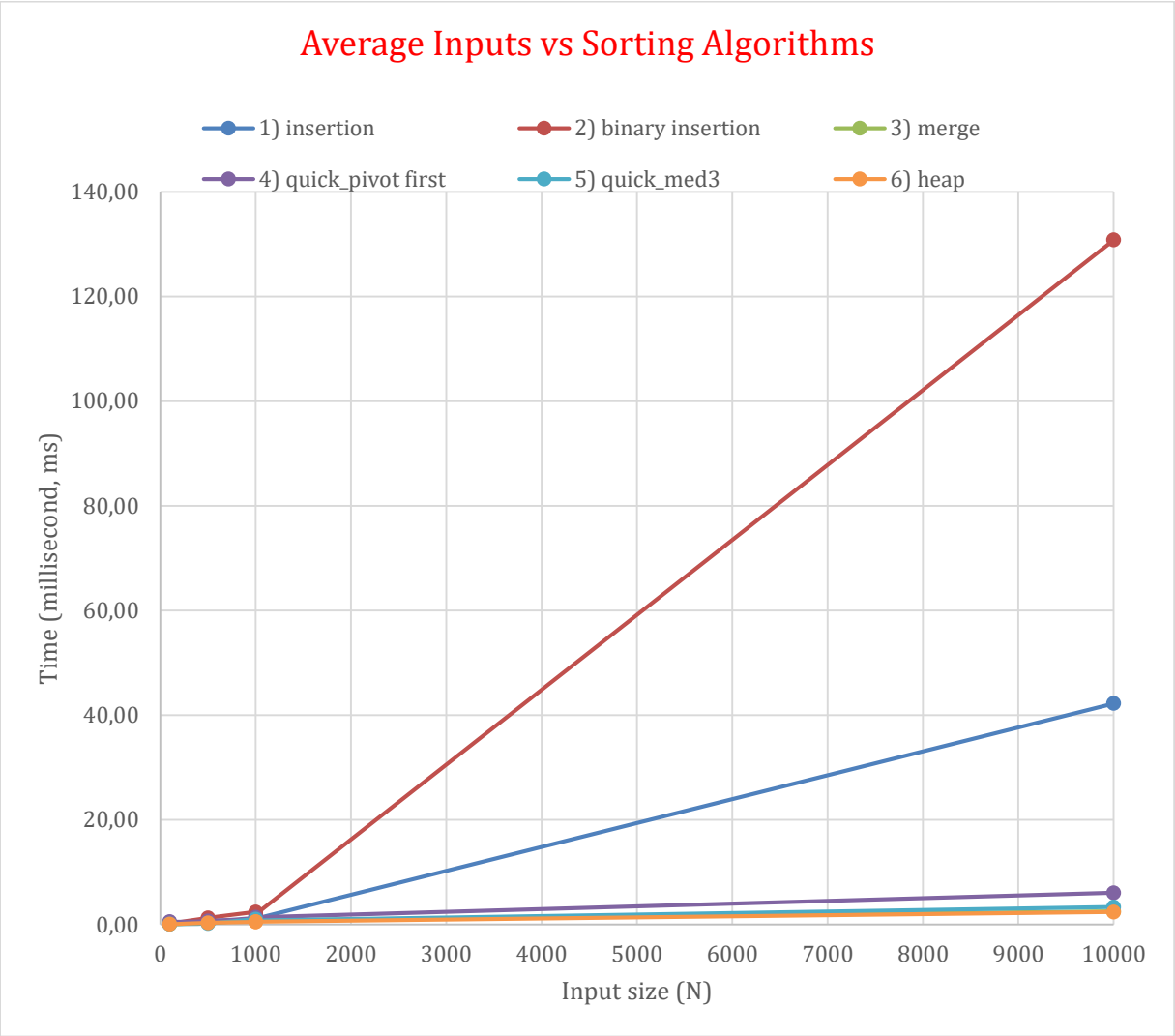

Figure 21 Average Inputs vs Sorting Algorithms

According to this graph, the worst algorithms for average cases are binary insertion and insertion algorithms. Insertion sort algorithms are more efficient if the input size is small, not appropriate for large arrays.

References:

- Levitin, “Introduction to Design and Analysis of Algorithms”, 3/e, Pearson
- https://en.wikipedia.org/wiki/Insertion_sort
- https://www.geeksforgeeks.org/merge-sort/
- https://www.geeksforgeeks.org/find-a-permutation-that-causes-worst-case-of-merge-sort/
- https://stackoverflow.com/questions/23919017/how-to-make-worst-case-permutation-of-median-of-three-quicksort
- https://en.wikipedia.org/wiki/Heapsort
- https://medium.com/basecs/counting-linearly-with-counting-sort-cd8516ae09b3