

MARMARA UNIVERSITY

**FACULTY OF ENGINEERING DEPARTMENT OF COMPUTER SCIENCE ENGINEERING**

**CSE 2046 – ANALYSIS OF ALGORITHMS COMPARING SORTING ALGORITHMS**

|  |  |
| --- | --- |
| **STUDENTS** | **NUMBERS** |
| Serkan Koç | 150118073 |
| Atila İlhan Yatağan | 150118033 |
| İbrahim Hakkı Candan | 150118061 |

|  |  |
| --- | --- |
| **Submitted To:** | **Due Date:** |
| Asst.Prof.Ömer Korçak | 11/05/2022 |

Contents

[Purpose of the Project 3](#_Toc102682326)

[Generating Inputs 3](#_Toc102682327)

[1. Insertion Sort: 3](#_Toc102682328)

[2. Merge Sort: 3](#_Toc102682329)

[3. Quick Sort (pivot is always selected as the first element): 4](#_Toc102682330)

[4. Partial Selection Sort: 4](#_Toc102682331)

[5. Partial Heap Sort: 4](#_Toc102682332)

[6. Quick Select Algorithm (pivot is always selected as the first element) 5](#_Toc102682333)

[7. Quick Select Algorithm (with median-of-three pivot selection) 5](#_Toc102682334)

[Deciding on Metrics 5](#_Toc102682335)

[Analyzing Results 5](#_Toc102682336)

[1. Insertion Sort: 5](#_Toc102682337)

[2. Merge Sort: 10](#_Toc102682338)

[3. Quick Sort (pivot is always selected as the first element): 12](#_Toc102682339)

[Comparing All Algorithms 16](#_Toc102682340)

[References: 17](#_Toc102682341)

# 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

## 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(n2).

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(n2).

## 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 2n elements.

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

sizes.

## 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(n2).

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

## Partial Selection Sort:

The concept used in Selection Sort helps us to partially sort the array up to kth smallest (or largest) element for finding the kth smallest (or largest) element in an array.

The best case is a sorted array in ascending order and k equals 1. The time complexity of the best case is O(n).

The worst case is a reversed sorted array in descending order and k equals n for partial selection sort. 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(nk).

## Partial Heap Sort:

## Quick Select Algorithm (pivot is always selected as the first element)

## Quick Select Algorithm (with median-of-three pivot selection)

# 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 ‘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

## 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*



Insertion Sort Best Case

Empirical

Theoretical

0,2500

0,2000

0,1500

0,1000

0,0500

0,0000

0

2000

4000

6000

8000

10000

Input size (N)

Time (millisecond, ms)

For the average case, we used random inputs that we generated before. For this case, the time complexity is O(n2). The amount of the time measured is increased about to n2 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*



Insertion Sort Average Case

Empirical

Theoretical

50,0000

40,0000

30,0000

20,0000

10,0000

0,0000

0

2000

4000

6000

8000

10000

Input size (N)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 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*

Time (millisecond, ms)

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*



Insertion Sort Worst Case

Empirical

Theoretical

80,0000

60,0000

40,0000

20,0000

0,0000

0

2000

4000

6000

8000

10000

Input size (N)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 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*

Time (millisecond, ms)

## 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*



Merge Sort Best Case

Empirical

Theoretical

1,4000

1,2000

1,0000

0,8000

0,6000

0,4000

0,2000

0,0000

0

2000

4000

6000

8000

10000

Input size (N)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 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*

Time (millisecond, ms)

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*



Merge Sort Average Case

Empirical

Theoretical

3,0000

2,5000

2,0000

1,5000

1,0000

0,5000

0,0000

0

2000

4000

6000

8000

10000

Input size (N)

Time (millisecond, ms)

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*



Merge Sort Worst Case

Empirical

Theoretical

4,5000

3,0000

1,5000

0,0000

0

2000

4000

6000

8000

10000

Input size (N)

Time (millisecond, ms)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 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*

## 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)*



Quick Sort Best Case

Empirical

Theoretical

6,0000

5,0000

4,0000

3,0000

2,0000

1,0000

0,0000

0

2000

4000

6000

8000

10000

Input size (N)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 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*

Time (millisecond, ms)

For the average case, we used random inputs that we generated before. For this case, the time complexity is O(n2). The amount of the time measured is increased about to n2 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*



Quick Sort Average Case

Empirical

Theoretical

8,0000

6,0000

4,0000

2,0000

0,0000

0

2000

4000

6000

8000

10000

Input size (N)

Time (millisecond, ms)

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 n2 times while input size is enlarging. Results coincide with time complexity O(n2).

*Table 12 Quick Sort Worst Case (first-pivot)*



Quick Sort First-Pivot Worst Case

Empirical

Theoretical

14,0000

13,0000

12,0000

11,0000

10,0000

9,0000

8,0000

7,0000

6,0000

0

2000

4000

6000

8000

10000

Input size (N)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 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*

Time (millisecond, ms)

# Comparing All Algorithms



Average Inputs vs Sorting Algorithms

1) insertion

140,00

4) quick\_pivot first

2) binary insertion

5) quick\_med3

3) merge

6) heap

120,00

100,00

80,00

60,00

40,00

20,00

0,00

0

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Input size (N)

Time (millisecond, ms)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

*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-](https://www.geeksforgeeks.org/find-a-permutation-that-causes-worst-case-of-merge-sort/) [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-](https://stackoverflow.com/questions/23919017/how-to-make-worst-case-permutation-of-median-of-three-quicksort) [permutation-of-median-of-three-quicksort](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