ALGORITHMS AND DATA STRUCTURES

UTRECHT UNIVERSITY

Two Timing Experiments

Author:
Mehrad Haghshenas (Student Number: 2822865)

Date:
October 5, 2023

This is all my own work. I have not knowingly allowed others to copy my work. This work has not been submitted for assessment in any other context.

1 Introduction

This project aims to investigate the performance of two distinct sorting algorithms through two separate experiments. Accordingly, the experiments will be on *selection sort* and *merge sort* algorithms. Java and Python have been used for programming the algorithms. The codes for the basis of the comparison (in Java) and for creating the graphs (in Python) has been written by the author. The code for selection sort (in Java) has also been written entirely by the author[1]; nevertheless, the code for the merge sort has been taken from the *geeksforgeeks* website [1]. The sorting algorithms will be compared on the basis of their performance, i.e., their time complexity when sorting data sets.
The experiments are conducted by executing each algorithm *a fixed number of times*; specifically, five thousand times. The elapsed time is recorded as following:

1. Five-hundred different arrays are created.

2. For each of which we run the two sorting algorithms ten times.

3. For each execution, the time that it takes for the sorting algorithm to sort the array is measured.

4. The average of the ten measurements is calculated. This will be the ultimate recorded time for sorting the corresponding array using the specified sorting algorithm.

The Big-Oh notation for the algorithms under investigation is also provided.
Experiment 1 focuses on comparing *selection sort* and *merge sort* using random permutations of data. What we mean by *random* is that the arrays are not sorted in any order - they are not saw-tooth in shape nor are they sorted in an ascending or descending order. *Permutations* of data means that the arrays do not contain duplicate elements. Experiment 2 assesses the same algorithms with data already sorted in ascending order that allow duplicate values. The null hypothesis for both experiments is formulated as the following: **There is no significant difference in the execution times of the two sorting algorithms**. In other words, the null hypothesis says that the mean of the elapsed time for multiple executions of the two algorithms is equal, indicating that any observed variations in execution times are purely due to random chance. The alternative hypothesis in both experiments will therefore be **there is a significant difference in the execution times between the two sorting algorithms**. The results are presented through box plots and graphs. The acceptance or rejection of the null hypothesis are tested and conclusions are provided. Last but not least, all numbers representing the execution times are in *nano seconds*.

[1]The reason for writing these code from scratch was to get more exercise in implementing these algorithms myself.

2 Experimental platform

The *hardware* used for the experiments was a macOS Ventura (Version 13.1) with 16 GB of RAM and an Apple M1 chip. The Apple's M1 chip includes 8 CPU cores. The software used was TMCBeans version 11.1. TMCBeans is similar to Netbeans with an auxiliary Test My Code plugin. Finally, the algorithms are single-threaded; hence, only one core will be used for the execution of the programs. However, note that TMCBeans might cause some overhead which has been exempted from our analysis.

3 Experiment 1

We will first state the complete *null hypothesis* and the *alternative hypothesis* for experiment 1. The null hypothesis states that no significant difference exists in the average execution times between the selection sort and merge sort algorithms when sorting *random permutations* of data.[2] The alternative hypothesis states that there is a significant difference in the execution times between the selection sort and merge sort algorithms when sorting random permutations of data. In other words, the null hypothesis says that the mean of the elapsed time for multiple executions of the two algorithms is equal, indicating that any observed variations in execution times are due to random chance. However, the alternative hypothesis says that the mean elapsed time between the two algorithms is significant. We will use a *t-test* to examine our null hypothesis and the significance level, ρ, will be *0.05*.
The experiment will time the sorting algorithms by running each algorithm on multiple random permutation arrays of varying sizes. Specifically, five-hundred different arrays are created; for each of which we run the two sorting algorithms ten times. For each execution, the time that it takes for the sorting algorithm to sort the array is measured. The average of the ten measurements is calculated. This will be the ultimate recorded time for sorting the corresponding array using the sorting algorithm. The reason that we calculate the mean of the ten iterations instead of just running the algorithm once on each array is to cancel out any unexpected behaviour. Lastly different array sizes will be used to assess the scalability of the algorithms. To be precise the arrays will have a length of 5 to 2500 incrementing by 5 in size.
The Big Oh notation for the two algorithms is as follows: Selection Sort: $O(n^2)$ - In other words the time complexity of selection sort is quadratic with respect to its input size. Merge Sort: $O(nlog_n)$.
Comparing the Big Oh of merge sort and selection sort, merge sort is expected to perform better with larger input sizes. By comparing the empirical execution times, this experiment aims to determine whether this theory is true; i.e., whether the better theoretical time complexity of merge sort will lead to it outperforming selection sort, leading

[2]We re-emphasize that in this experiment the arrays do not contain duplicate elements and they are not sorted in any order.

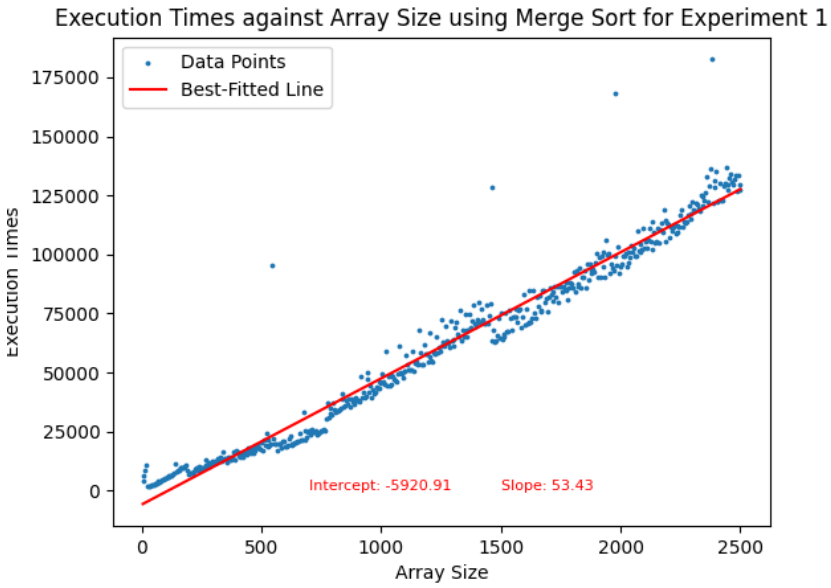

Figure 1: execution time against array size for the merge sort

Selection	=	[2412,	4770,	1808,	2562,	3545,	4487,	5637,	6700,
		7970,	9058,	10670,	12937,	12150,	5195,	2491,	2800,
		3258,	3283,	3475,	3849,	4145,	4854,	5008,	5120]

Table 1: First twenty execution times of the Selection sort algorithm.

to the rejection of the null hypothesis.
The following are the scatter plot of execution time against array size for the merge sort and selection sort respectively.
As it can be seen the execution time of the merge sort against array size follows a much more linear path compared to selection sort. In other words, the merge sort has a less *growth step* whereas the selection sort has a polynomial growth rate. The intercepts and the slope of the best fitted lines for each of the scatter plots can be seen on the figures. A said before, we emphasize that the points on the graph represent mean values as the sorting algorithms were executed ten times for each array and the mean was calculated. The box plot of the two algorithms can be seen in figure 3 and 4.
Unfortunately the complete data table of the timing results cannot be provided given that for each algorithm, there are 500 data inputs. The reader can refer to the Python codes to see the full data inputs. However, for the sake of completeness, I have put the first twenty mean execution times of each algorithm which can be seen in tables 1 and 2.
To test the validity of the null hypothesis - no significant difference exists in the average execution times between the selection sort and merge sort algorithms when sorting

Figure 2: execution time against array size for the selection sort

Figure 3: box plot of execution time of merge sort

Merge	=	[3833,	6183,	8558,	10637,	1741,	1595,	1753,	2233,
		2408,	2362,	3016,	3008,	3245,	3941,	3725,	4104,
		4620,	4741,	5112,	5299,	5683,	6050,	6154,	6325]

Table 2: First twenty execution times of the Merge sort algorithm.

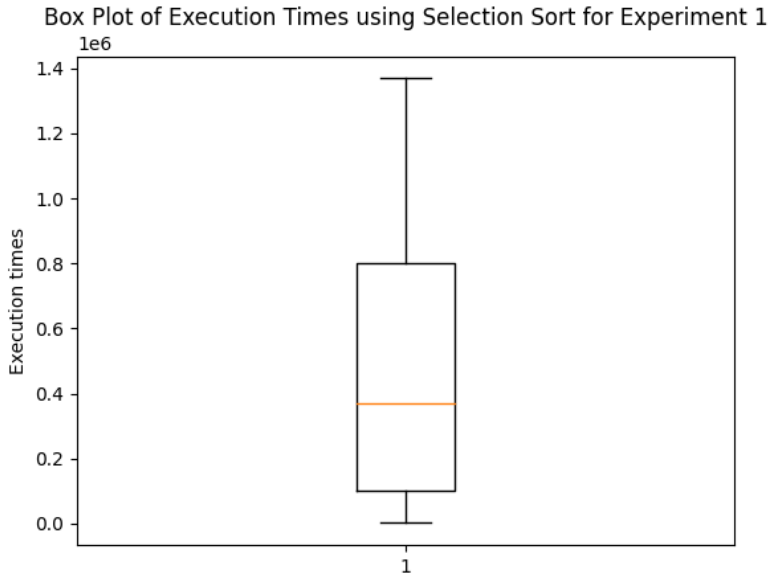

Figure 4: box plot of execution time of selection sort

random permutations of data - we do a *two tailed t-test*. A two tailed test is carried out rather than a one-tailed because the alternative hypothesis is of the form $\mu_1 \neq \mu_2$. The two tailed t-test has been carried out in python with a significance level of 0.05. The p-value was $4.17 \times (10^{-91})$. Therefore, the null hypothesis is rejected, and the means are significantly different. In other words, there is a significance difference between the execution time of merge sort and selection sort. This practical result completely aligns with the Big Oh values of the two algorithms previously mentioned. As a final note, the median and mean of the execution times for the selection sort are 369716.5 and 478203.494 respectively. For merge sort, these numbers are 62537.0 and 60995.5 in order.
The code fragments that are to be compared are methods selectionSort (lines 86-94) and mergeSort (lines 171-183) of the Experiment1.java file.

```
1 public static void selectionSort(int[] arr) {
2     int temp;
3     for (int i = 0; i < arr.length; i++) {
4         temp = arr[i];
5         arr[i] = arr[minimum(i, arr)];
6         arr[minimum(i, arr)] = temp;
7     }
8 }
```

In selection sort, for each element α of the array we *swap* it with the minimum element of the array after α. This is what the loop does.

```
1 public static void mergeSort(int arr[], int l, int r) {
```

```
    if (l < r) {
        // Find the middle point
        int m = (l + r) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}
```

In Merge sort, we recursively break the array and sort the broken pieces and then merge them back to originate the sorted version of the initial array. It a sense, we start from sorting the leaves of a tree and go up to the root node; where the nodes of the tree are sub-partitions of the array.

4 Experiment 2

Experiment 2 is very similar to experiment 1 in structure; meaning that the assumptions in the previous section hold for experiment 2. Thus, for the sake of conciseness, we will not repeat them here. The null hypothesis for Experiment 2 is that there is no significant difference in the execution times of selection sort and merge sort when sorting data that is already sorted in ascending order. Note that the data *can* contain duplicate values. The alternative hypothesis is that there is a significant difference in the execution times. The timing *for each execution* was performed by subtracting the start time of the execution from the end time. The timing *for each array* was averaging the ten times allocated for a specific array. The Big Oh is still the same as experiment 1. Figure 5 and 6 are the scatter plot of execution time against array size for the merge sort and selection sort respectively.
Again, as it can be seen the execution time of the merge sort against array size follows a much more linear path compared to selection sort. In other words, the merge sort has a less *growth step* whereas the selection sort has a polynomial growth rate. The intercepts and the slope of the best fitted lines for each of the scatter plots can be seen on the figures. A said before, we emphasize that the points on the graph represent mean values as the sorting algorithms were executed ten times for each array and the mean was calculated.
The box plot of the two algorithms can be seen in figures 7 and 8.
Unfortunately the complete data table of the timing results cannot be provided given that for each algorithm. I have put the first twenty execution times of each algorithm in tables 3 and 4.
To test the validity of the null hypothesis - no significant difference exists in the average execution times between the selection sort and merge sort algorithms when sorting

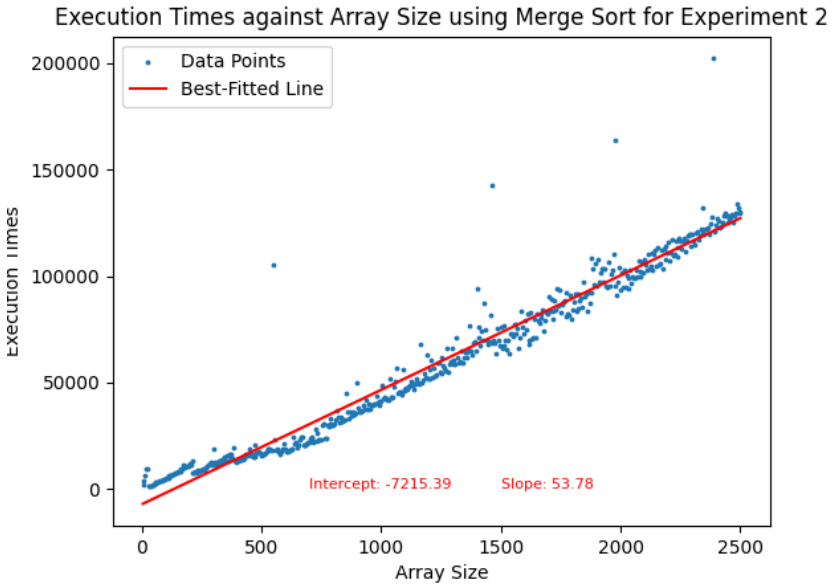

Figure 5: execution time against array size for the merge sort

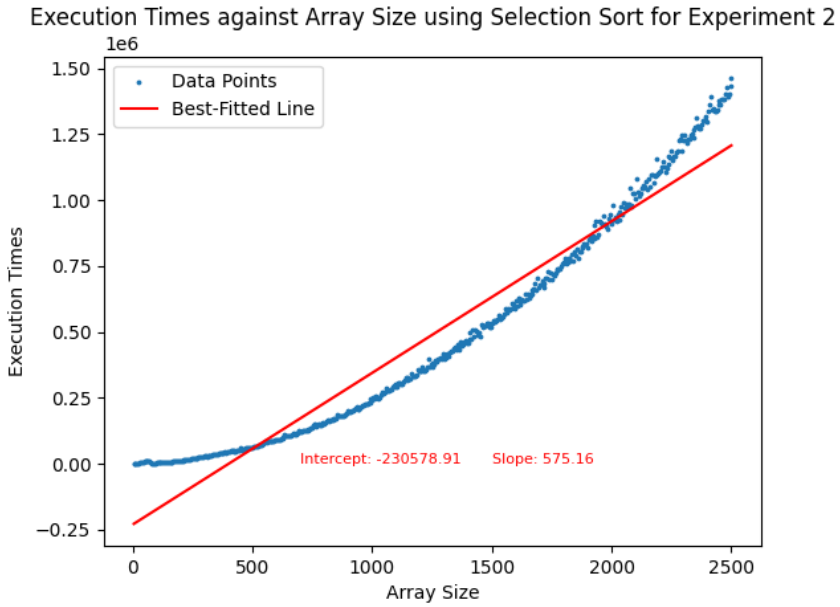

Figure 6: execution time against array size for the selection sort

Figure 7: box plot of execution time of merge sort

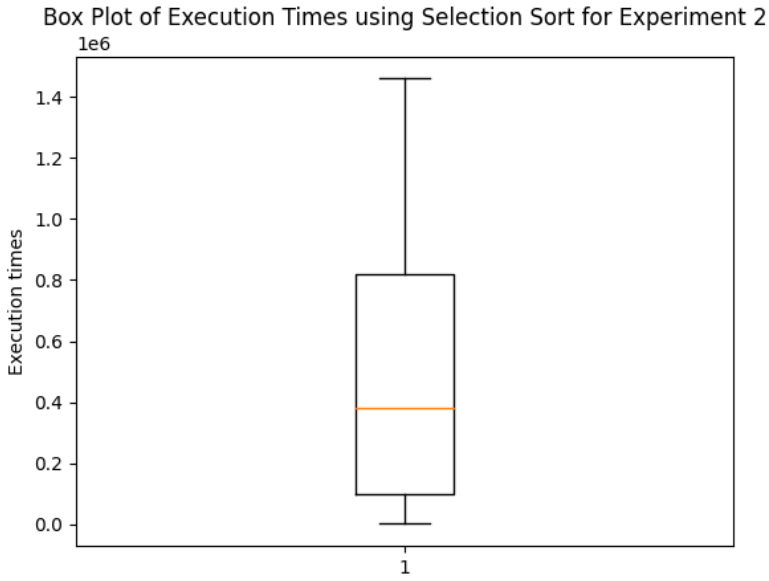

Figure 8: box plot of execution time of selection sort

Selection	=	[933,	2583,	2858,	1716,	2429,	3454,	4299,	5254,
		6574,	8058,	9641,	8658,	10362,	11720,	5220,	2204,
		2554,	2608,	2837,	3112,	3262,	3887,	4341,	4491,
		4858,	5000,	5366,	5700,	6658,	6829,	7345,	7400,

Table 3: First twenty execution times of the Selection sort algorithm.

Merge	=	[3833,	6183,	8558,	10637,	1741,	1595,	1753,	2233,
		2408,	2362,	3016,	3008,	3245,	3941,	3725,	4104,
		4620,	4741,	5112,	5299,	5683,	6050,	6154,	6325

Table 4: First twenty execution times of the Merge sort algorithm.

already sort (in ascending order) of data, we do a *two tailed t-test*. A two tailed test is carried out rather than a one-tailed because the alternative hypothesis is of the form $\mu_1 \neq \mu_2$. The two tailed t-test has been carried out in python with a significance level of 0.05. The p-value was $4.19 \times (10^{-90})$. Although, the p-value is bigger than in experiment one, it is still very small and therefore, the null hypothesis is rejected, and the means are significantly different. As a final note, the median and mean of the execution times for the selection sort are 380051.5 and 489812.884 respectively. For merge sort, these numbers are 58843.5 and 60141.656 in order. If you compare these numbers with the numbers in experiment 1, they are very close, emphasizing that the status of the arrays are not important when the sorting algorithms are carried out.
The code fragments that are to be compared are methods selectionSort (lines 85-93) and mergeSort (lines 170-182) of the Experiment2.java file. The methods are exactly the same as experiment 1, so there is no need to write the methods again.

5 Conclusion

The reason to carry out these two experiments was to compare the performance of the merge and selection sorting algorithms. We found out that the practical experiments align with the theoretical definitions of the Big Oh values. Moreover, by carrying out these two experiments we found out that the status of the arrays does not impact the performance of the algorithms. In other words, whether the array was sorted or it did not follow a specific pattern, either way, the sorting algorithms roughly gave back the same number for the execution time. This is quite un-intuitive given that we expect that at least when the arrays are sorted then the algorithms should sort the array much faster because there is less work to do. Likewise, we expect that the selection sort and the merge sort have similar performances, time complexity wise, when the array is sorted because there is no *job* to be done. However, the algorithms still do a full analysis regardless of the status of the arrays. I believe a more *lazy approach* might be more practical. Nonetheless, the results of this experiment align with the Big Oh values

of the two algorithms previously mentioned.
Ultimately, one further analysis was carried out: the p-value of the execution time against the size of the array was calculated. This number for the selection sort in experiment 1 was 3.9×10^{-317}. For experiment 2, it was $1.87 \times 10{-}312$. The p-value of the execution time against the size of the array for the merge sort in both experiments was 0. All these numbers show that there is a *significant* relationship between the execution time and the size of the array, which was expected as well. This means that the difference between the execution times when increasing the array size is not due by chance.

References

[1] GeeksforGeeks. Java program for merge sort. Accessed: September 27, 2023. pages 2