STA2005S ED Assignment

Manelisi Luthuli (LTHMAN009) and Zenande Mangcipu (MNGZEN008)

2024-09-16

Objectives, Sources of Variation, and Model Used

The primary objective of this experiment is to compare the efficiency of various sorting algorithms by measuring the time they take to sort randomly generated data sets within different sized arrays. This will allow for an understanding of how each algorithm performs under different conditions, specifically when working with varying data set sizes. The performance of the algorithms will be analysed, and the execution time captured.

Several sources of variation are considered in this experiment. The treatment factor is the different sorting algorithms with 3 levels, namely Bubblesort, Heapsort, and Countsort. The experimental units are the arrays containing randomly generated data sets. They will be sorted by each algorithm. These arrays will vary in size to simulate different conditions and ensure that the performance of the algorithms is not dependent on the specific characteristics of one particular array.

The main blocking factor in this experiment is the size of the arrays. Array size has a direct impact on the time complexity and performance of each sorting algorithm, and thus needs to be controlled. The levels of the blocking factor are defined as 50,000, 100,000, 200,000, and 400,000 sized arrays. By accounting for these sources of variation, the experiment aims to provide a fair and comprehensive comparison of the efficiency of the selected sorting algorithms.

The experiment will use a randomized block design to compare and analyse the results.

Randomization Procedure, Data Collection/Creation, and How the Experiment Was Run

A "class" is a program that contains methods that can be used in other programs (classes). The experiment was run in the `Data` class in Java, but helper classes such as `ArrayGeneration`, `RandomArrayIndexes`, and `Shuffler` provided essential tools that assisted in carrying out the experiment. As a result of blocking according to the size of the arrays, 9 arrays were created for each block. Therefore, there were 9 arrays of size 50,000, 9 arrays of size 100,000, 9 arrays of size 200,000, and 9 arrays of size 400,000. This was to ensure repetition, as each algorithm was to act on 3 arrays of the same size within each block. In total, there were 36 arrays which were the experimental units.

The `ArrayGeneration` class was responsible for creating the arrays using two true random number generators: `/dev/urandom` and the Random.org Application Interface. The method `fetchRandomNumbersFromURandom` generated random data directly from `/dev/urandom`, a special file in Unix-like operating systems. This file provided access to entropy collected from hardware-based events like keyboard timings, mouse movements, or disk I/O. These unpredictable physical phenomena ensured the randomness of the numbers generated and stored within the arrays.

Furthermore, the Random.org Application Interface, on the other hand, was another source of true randomness that was used in the experiment. It generated random numbers by measuring atmospheric noise—specifically, variations in radio signals caused by unpredictable atmospheric conditions. The `RandomArrayIndexes` class used the Random.org Application Interface for each block to generate a unique (with no repetition of numbers)

random sequence of indexes from 0 to 8. The `Shuffler` class then shuffled the initial sequence of the indexes of the 9 arrays within each block to new positions that corresponded to the random sequence that was generated by the `RandomArrayIndexes` class. This ensured that the order of the arrays was randomized for each experimental block. The shuffling process utilized the method `shuffleArrayList`, which rearranged the arrays based on the random indexes fetched from the Random.org Application Interface.

After the above step, the `RandomArrayIndexes` class was utilized again to generate a sequence of unique random indexes from 0 to 35, corresponding to all the experimental units. These were then stored as values in an array. We then looped through the array that contained the sequence of unique random indexes from 0 to 35, and each index value was used to select a specific array (experimental unit). A sorting algorithm, which is the treatment, was then assigned based on the index of the array. This treatment was determined by the function `getSortingAlgorithm(index)` in the `Data` class, which mapped different indexes to specific sorting algorithms. For instance, indexes that were multiples of 3 were assigned Bubblesort, indexes that were one less than the multiples of 3 were assigned Heapsort, and the remaining indexes were assigned Quicksort. This approach ensured that treatments were distributed in a randomized manner rather than being sequentially applied to the arrays.

Furthermore, the index also dictated the order in which the arrays were processed. Rather than always processing the arrays in the same order (e.g., first sorting out all the 50,000-sized arrays, then the 100,000-sized ones, and so on), the index determined randomly which array was sorted next. Therefore, the code ensured that the arrays of various sizes were processed in a randomized order controlled by the randomized index.

- If the index fell between 0 and 8, it selected an array from 50,000.
- If the index fell between 9 and 17, it selected an array from 100,000.
- If the index fell between 18 and 26, it selected an array from 200,000.
- If the index fell between 26 and 35, it selected an array from 400,000.

This entire process ensured that when the experiment was run, the experimental units with their assigned treatments were processed in a non-sequential, randomized order from the Random.org Application Interface, preventing any potential bias from arising due to the sequence in which arrays of different sizes were sorted.

After each treatment (sorting algorithm) was applied to an array, the time taken to sort the array was recorded in an Excel sheet.

Difficulties Encountered During the Experiment

It was challenging to find true random generators and not pseudo-random generators. Also, there were cost constraints that arose on some of the generators. To work around these, we made use of `/dev/urandom`. We also utilized Random.org's free trial and ensured that we did not exceed the request limit for true random numbers. Additionally, coding the experiment in Java was also quite challenging.

Randomized Block Design Model

In a Randomized Block Design (RBD), the model equation is:

$$Y_{ij} = \mu + \tau_i + \beta_j + \epsilon_{ij}$$

where:

- Y_{ij} is the observed response for the i-th treatment in the j-th block.
- μ is the overall mean of the response.
- τ_i is the effect of the i-th treatment.
- β_j is the effect of the j-th block.
- ϵ_{ij} is the random error associated with the i-th treatment in the j-th block.

Explanation

1. **Overall Mean (μ)**: Represents the average response across all treatments and blocks.
2. **Treatment Effect (τ_i)**: Measures the deviation of the i-th treatment from the overall mean.
3. **Block Effect (β_j)**: Accounts for the variability among blocks.
4. **Error Term (ϵ_{ij})**: Represents the random variation not explained by the treatment or block effects.

The error term ϵ_{ij} is assumed to be normally distributed with mean 0 and constant variance σ^2. Mathematically, this is expressed as:

$$\epsilon_{ij} \sim \mathrm{N}(0, \sigma^2)$$

where:

- $\mathrm{N}(0, \sigma^2)$ denotes the normal distribution with mean 0 and variance σ^2.
- σ^2 is the constant variance of the error term.

Analysis

We will now model our data using RBD and perform ANOVA tests.

Table 1: Detailed ANOVA Table for Random Block Design Model

Term	Df	Sum_Sq	Mean_Sq	F_value	Pr_F
Array_Size	3	22406495710	7468831903	5.004661	0.0062274
Sorting_Algorithm	2	31538747451	15769373725	10.566628	0.0003359
Residuals	30	44771256095	1492375203	NA	NA

The ANOVA results show two significant factors:
Array Size: $F(3, 30) = 5.005$, $p = 0.006$, indicating that different array sizes significantly impact the sorting time.
Sorting Algorithm: $F(2, 30) = 10.567$, $p = 0.000336$, showing a statistically significant difference between the algorithms.
Both the size of the array and the algorithm used to sort it have a statistically significant impact on sorting time. This suggests that as array sizes increase, some algorithms may become less efficient, and there are clear performance differences between sorting algorithms.

Table 2: Table 1: Means of the Sorting Algorithms

	x
BubbleSort	62805.66667
HeapSort	20.33333
QuickSort	14.83333

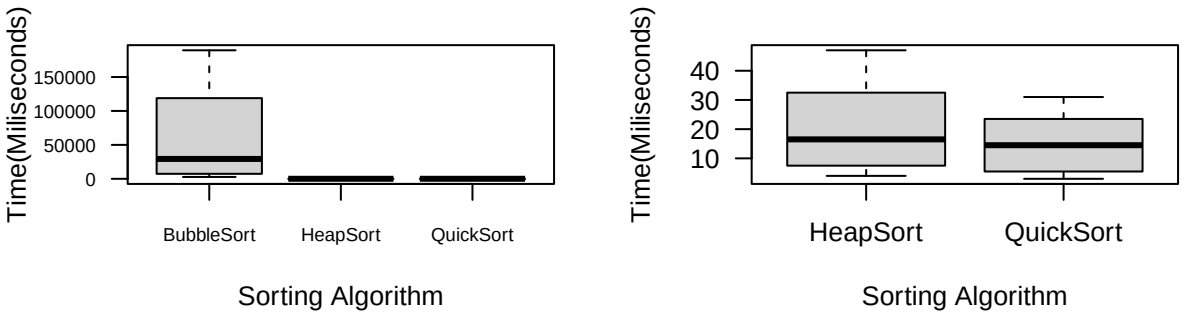
Table 3: Table 2: Means for the Different Array Sizes

	x
100000 elements	3955.5556
200000 elements	15824.4444
400000 elements	63069.5556
50000 elements	938.2222

Sorting Algorithms: Bubble Sort has a much higher mean sorting time (62,806 ms), while Heap Sort and Quick Sort are significantly faster (20 ms and 15 ms, respectively).

Array Sizes: Larger arrays (e.g., 400,000 elements) take much longer to sort compared to smaller ones (e.g., 50,000 elements).

Bubble Sort is consistently the slowest algorithm, while Heap Sort and Quick Sort are nearly identical in performance, with very small differences in sorting time. Sorting times increase significantly as array size grows, reflecting the expected increase in time complexity for sorting large datasets.



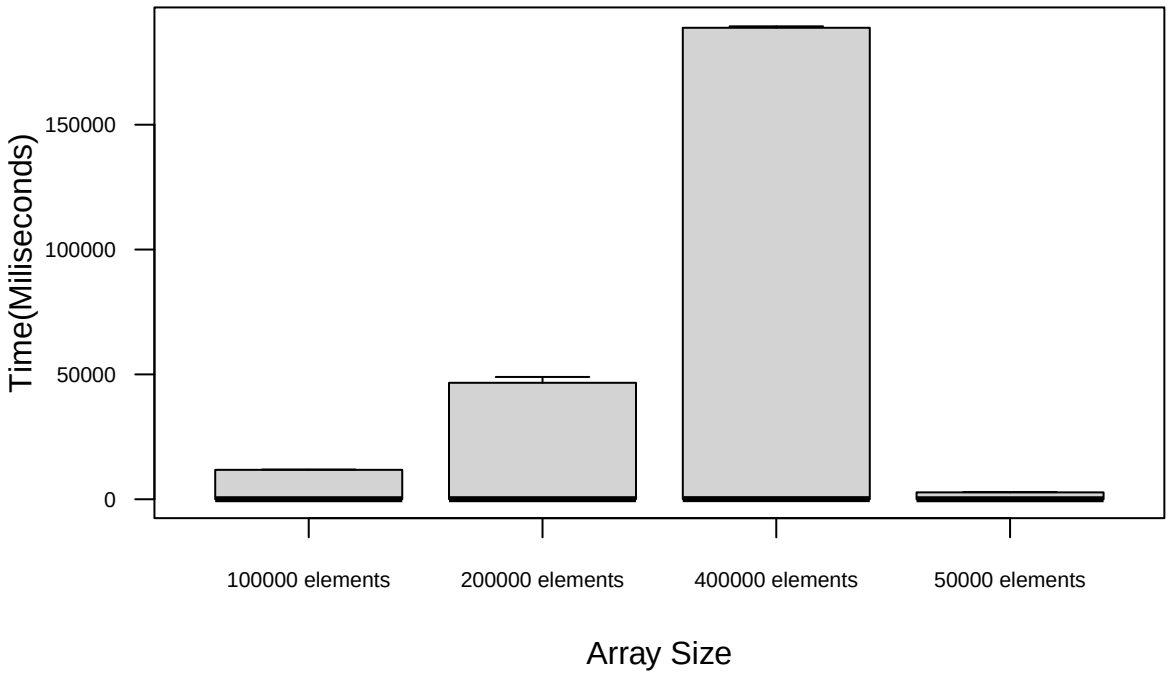
The box plots clearly show that Bubble Sort has much higher sorting times, with greater variability compared to Heap Sort and Quick Sort. Both Heap Sort and Quick Sort have lower, similar sorting times with very little variation.

Table 4: Pairwise Comparison Results with Bonferroni Adjustment at 5% signifance level

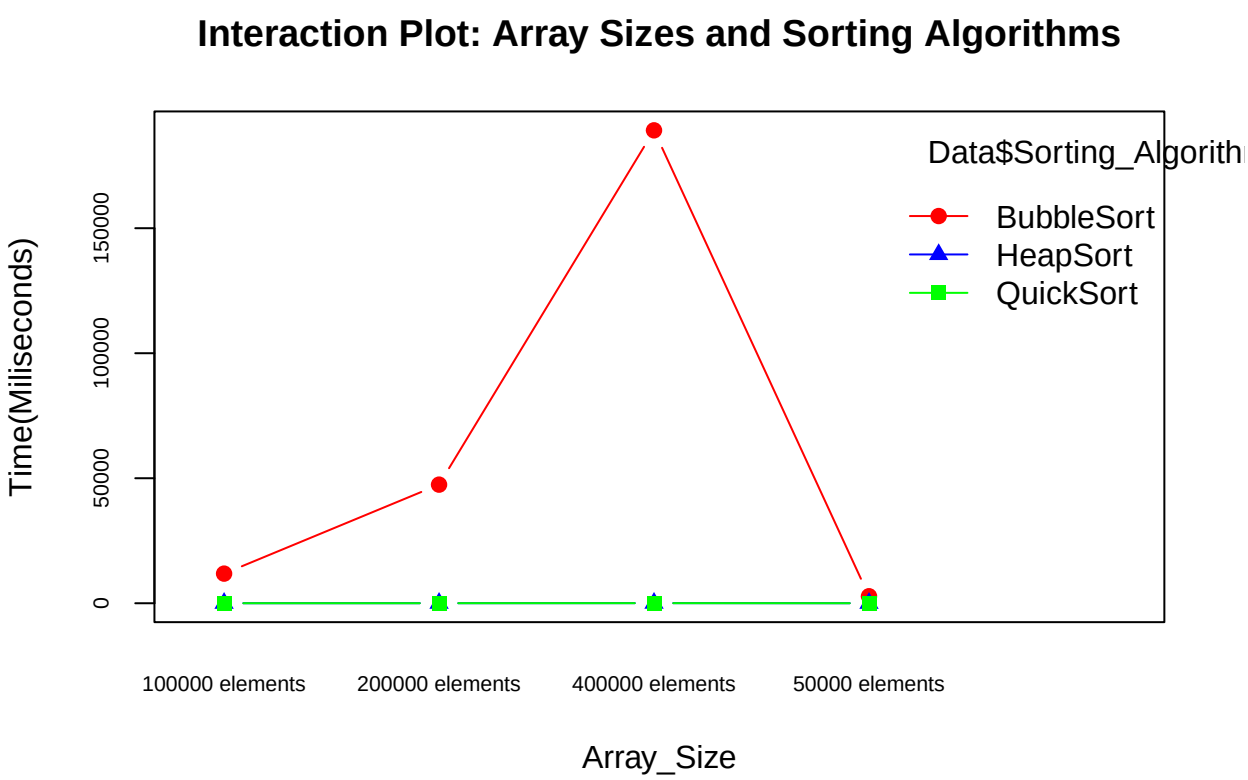
	BubbleSort	HeapSort
HeapSort	0.0052115	NA
QuickSort	0.0052073	1

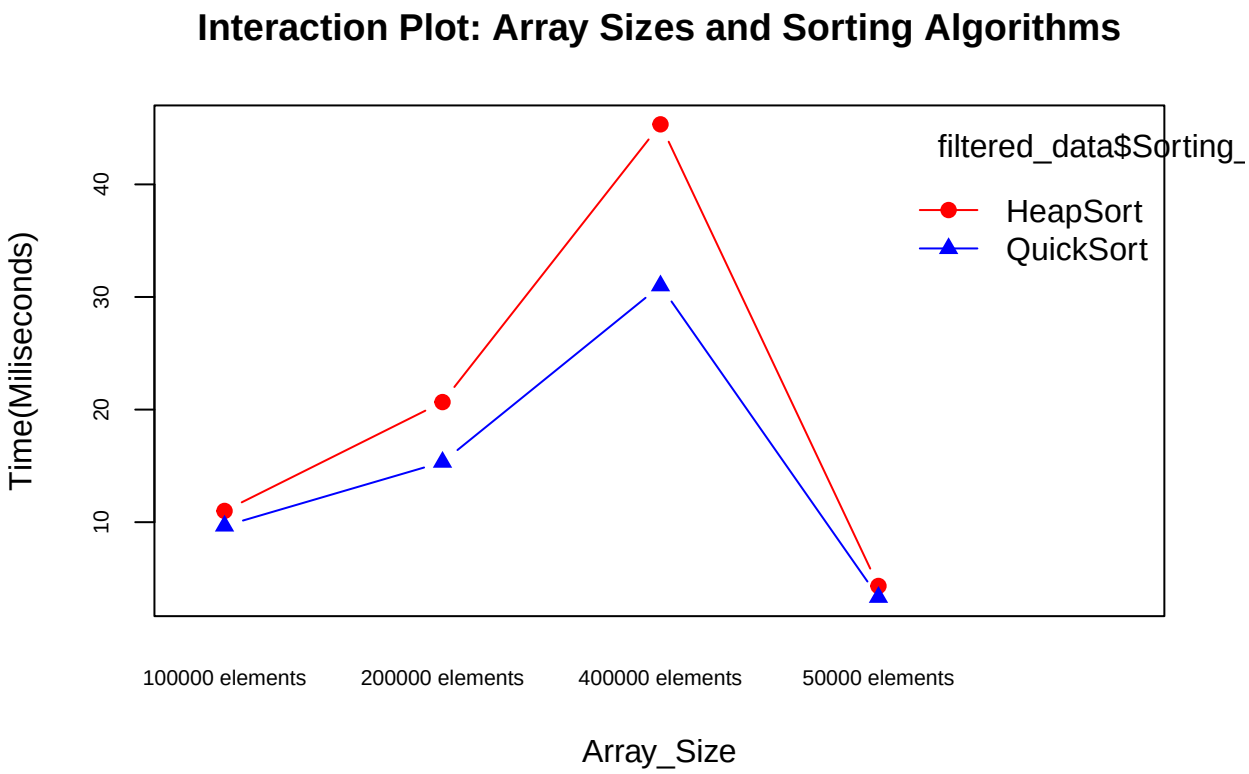
The Bonferroni-adjusted pairwise comparisons indicate significant differences between Bubble Sort and both Heap Sort and Quick Sort ($p = 0.0052$ for both). However, there is no significant difference between Heap Sort and Quick Sort ($p = 1.000$).

Bubble Sort is significantly slower than both Heap Sort and Quick Sort. Heap Sort and Quick Sort perform similarly, with no statistically significant difference in their efficiency.



The differences between the blocks is very high. Therefore the assumption of equal variances is not met for our model, and conclusions could be wrong.


Interaction Plot: Array Sizes and Sorting Algorithms
Time(Miliseconds)
0
50000
100000
150000
Data$Sorting_Algorith
BubbleSort
HeapSort
QuickSort
100000 elements
200000 elements
400000 elements
50000 elements
Array_Size

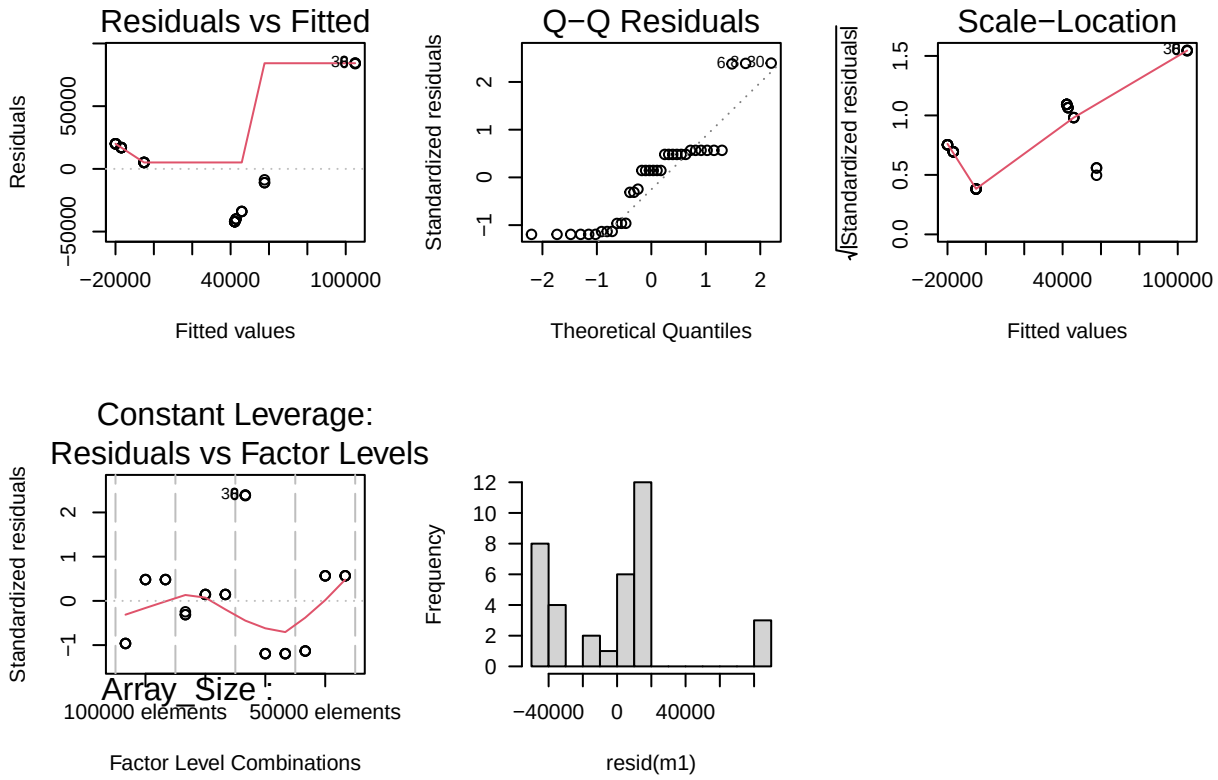


The lines for HeapSort and QuickSort appear to be relatively parallel across different array sizes, indicating no significant interaction between these algorithms and array sizes. However, BubbleSort shows a much steeper increase in sorting time as array size increases, suggesting that while there is a strong main effect of array size and sorting algorithm, there may not be a clear interaction between blocks (array sizes) and treatments (sorting algorithms) overall. Thus, blocks and treatments do not show a strong interaction, but the performance differences between algorithms become more pronounced with larger arrays. Therefore the assumption of no interactions for the model on our data has no evidence against it.

Table 5: Shapiro-Wilk Test for Normality of Residuals

	Statistic	p_value
W	0.8538687	0.0002339

Since the p-value is less than significance level of 0.001, we reject the null hypothesis of the Shapiro-Wilk test, which assumes that the residuals follow a normal distribution. This suggests that the residuals of the model are not normally distributed. Thus this means that our error term of RBD assumption fails.



Residuals vs. Fitted Plot: The plot shows potential patterns, suggesting that the assumption of homoscedasticity might be violated. This means that the variance of residuals could be changing with different fitted values, which could lead to unreliable results from the model.

Q-Q Plot: The Q-Q plot compares the distribution of the residuals to a theoretical normal distribution. Points should lie along the 45-degree line if the residuals are normally distributed. In this case, the points deviate from the line, especially in the tails, indicating that the residuals are not normally distributed. This confirms the result from the Shapiro-Wilk test ($W = 0.85387$, $p = 0.0002339$), which also suggested non-normality of residuals.

Scale-Location Plot: The Scale-Location plot (also called a Spread-Location plot) shows whether the residuals are spread equally across all levels of the fitted values. The plot should display a horizontal line with evenly scattered points if the variance of the residuals is constant. In this case, the plot shows some trend, indicating that the residuals are not evenly spread. This suggests heteroscedasticity, meaning that the residuals' variance changes with fitted values.

The diagnostic plots and the Shapiro-Wilk test indicate that the residuals are not normally distributed. There is evidence of heteroscedasticity, as the residuals do not have constant variance.

Conclusion

Violations of model assumptions (normality and homoscedasticity) suggest that the conclusions drawn from the ANOVA results in the RBD model may not be fully reliable. It may be necessary to explore transformations of the data or use a non-parametric approach to address these issues.

APPENDIX

```
## # A tibble: 36 x 3
##    Sorting_Algorithm Array_Size      `Time(Miliseconds)`
##    <chr>             <chr>                       <dbl>
##  1 BubbleSort        200000 elements             48975
##  2 QuickSort         100000 elements                15
##  3 HeapSort          100000 elements                13
##  4 QuickSort         50000 elements                  3
##  5 BubbleSort        100000 elements             11872
##  6 BubbleSort        400000 elements            188821
##  7 QuickSort         100000 elements                 7
##  8 BubbleSort        400000 elements            189186
##  9 HeapSort          400000 elements                47
## 10 QuickSort         50000 elements                  4
## # i 26 more rows
```

The below is a link to code that runs the experiment, it can be found in the "src" folder. The R markdown code which contains that R code for the analysis can also be found here named "trial.Rmd". The data from the experiment can also be found here named, "ExperimentalDesign.xlsx".

https://drive.google.com/drive/folders/1WJUbxh98JU9CQ8ee8euJrC_Jw6ThZQhV?usp=sharing