

**CO2412: Computational Thinking**

## Assignment

## Sort the Data!

## by

## Dinira Milanaka Pathirana

## UCLan ID: G21236306

## Submitted to: Dr. Madushi Pathmaperuma

## Word count: 1096

## January 22nd, 2024

Table of Contents

[I. Introduction 3](#__RefHeading___Toc705_1410167119)

[II. Big O Notation 3](#__RefHeading___Toc709_1410167119)

[a. Definition: 3](#__RefHeading___Toc711_1410167119)

[b. Common Time Complexities: 3](#__RefHeading___Toc715_1410167119)

[III. Sorting Algorithms and Big O Notation Analysis 4](#__RefHeading___Toc717_1410167119)

[a. Selection Sort: 4](#__RefHeading___Toc719_1410167119)

[b. Merge Sort: 5](#__RefHeading___Toc721_1410167119)

[c. Quick Sort 6](#__RefHeading___Toc723_1410167119)

[IV. Sorting Algorithm’s Performance Comparisons 7](#__RefHeading___Toc725_1410167119)

[a. Performance Comparisons 7](#__RefHeading___Toc1524_2042955154)

[Table 1: Array Size 100 7](#__RefHeading___Toc1360_2042955154)

[Table 2: Array Size 1000 8](#__RefHeading___Toc1362_2042955154)

[Table 3: Array Size 10000 9](#__RefHeading___Toc1364_2042955154)

[b. Identification of the Fastest Algorithm: 10](#__RefHeading___Toc739_1410167119)

[Table 4: Summarizing algorithm comparisons across array sizes: 10](#__RefHeading___Toc1366_2042955154)

[V. Conclusion 10](#__RefHeading___Toc741_1410167119)

[VI. References 10](#__RefHeading___Toc743_1410167119)

[VII. Bibliographies 11](#__RefHeading___Toc2070_1840836406)

Table of Figures

[Figure 1: Big O Notation Graph [11] 3](#Figure!0|sequence)

[Figure 2: Selection Sort Big O Notation Graph 4](#Figure!1|sequence)

[Figure 3: Merge Sort Big O Notation Graph 5](#Figure!2|sequence)

[Figure 4: Quick Sort Big O Notation Graph 6](#Figure!3|sequence)

[Figure 5: Graph for 100 elements 7](#Figure!4|sequence)

[Figure 6: Graph for 1000 elements 8](#Figure!5|sequence)

[Figure 7: Graph for 10000 elements 9](#Figure!6|sequence)

Index of Tables

Table 1: Array Size 100 6

Table 2: Array Size 1000 7

Table 3: Array Size 10000 8

Table 4: Summarizing algorithm comparisons across array sizes: 9

# I. Introduction

Sorting algorithms are crucial for handling larger datasets, directly impacting computer system performance. Our report focuses on Big O Notation to assess algorithm speed. We examine three sorting algorithms Selection Sort, Merge Sort, and Quick Sort to identify the fastest performer, aiming to enhance computer system functionality.

# II. Big O Notation

## a. Definition:

Big O Notation is a standardized method for expressing an algorithm's worst-case time complexity [2]. It provides a concise representation, allowing systematic assessment of efficiency. Significantly, it aids in comparing algorithms, helping select the most suitable for tasks, especially with large datasets. A lower Big O value indicates superior efficiency, making it a critical tool in computer science for informed algorithmic analysis and selection [8].

## b. Common Time Complexities:

* **O(1) - Constant Time Complexity:** The algorithm's speed remains the same, regardless of the data size.
* **O(log n) - Logarithmic Time Complexity:** The algorithm's speed increases slowly as the data size grows.
* **O(n) - Linear Time Complexity:** The algorithm's speed grows linearly with the data size.
* **O(n log n) - Logarithmic Time Compl**exity: The algorithm's speed increases, but not as fast as linear growth.
* **O(n^2) - Quadratic Time Complexity:** The algorithm's speed grows quadratically with the data size.

[**Figure 1**](#_toc12) depicts the graph illustrating the common complexities discussed above.

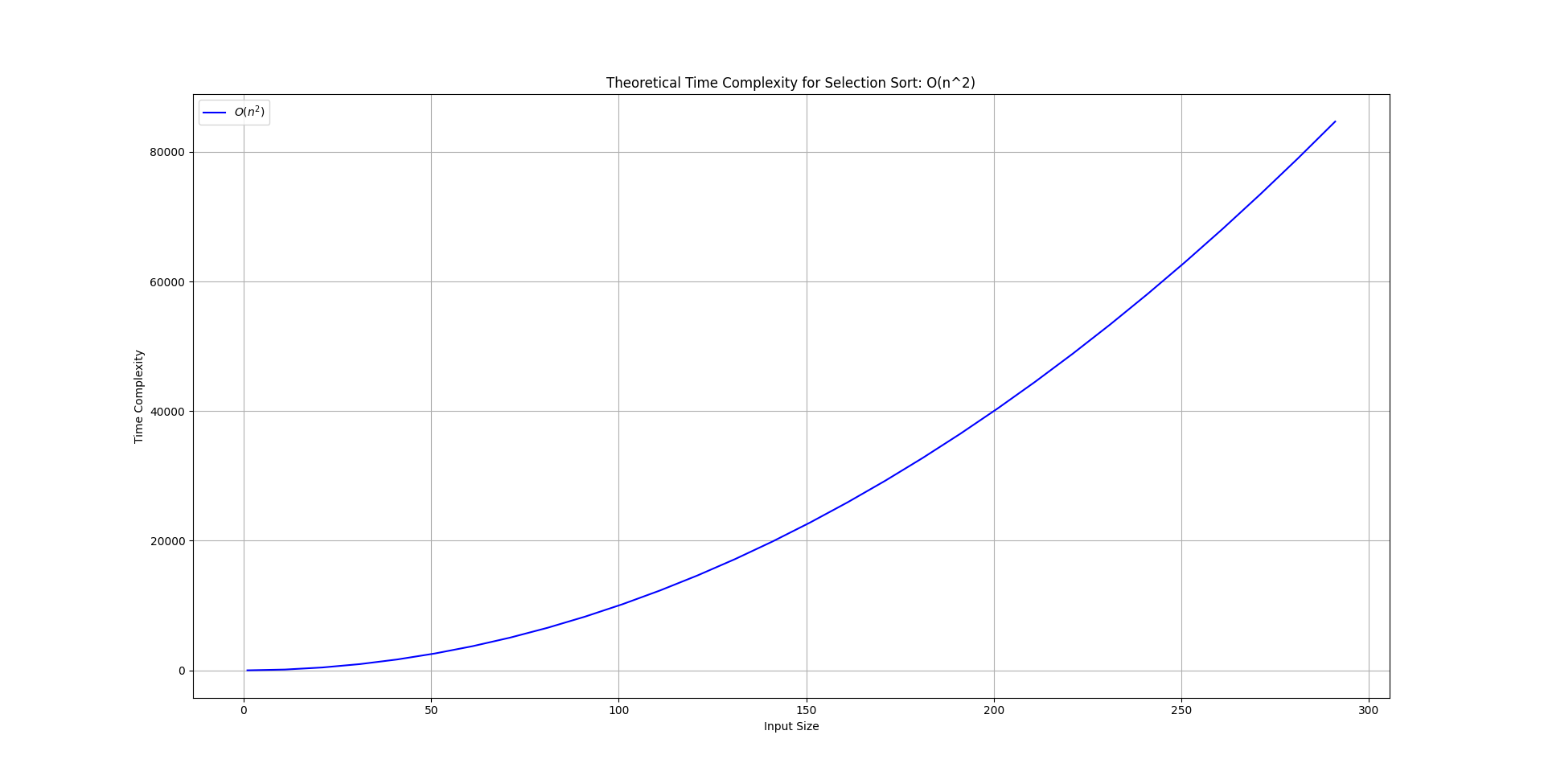
**Figure 1: Big O Notation Graph [11]**

# III. Sorting Algorithms and Big O Notation Analysis

## a. Selection Sort:

Selection Sort is a straightforward sorting algorithm that iteratively selects the minimum element from the unsorted part of the array and places it at the beginning until the entire array is sorted [10].

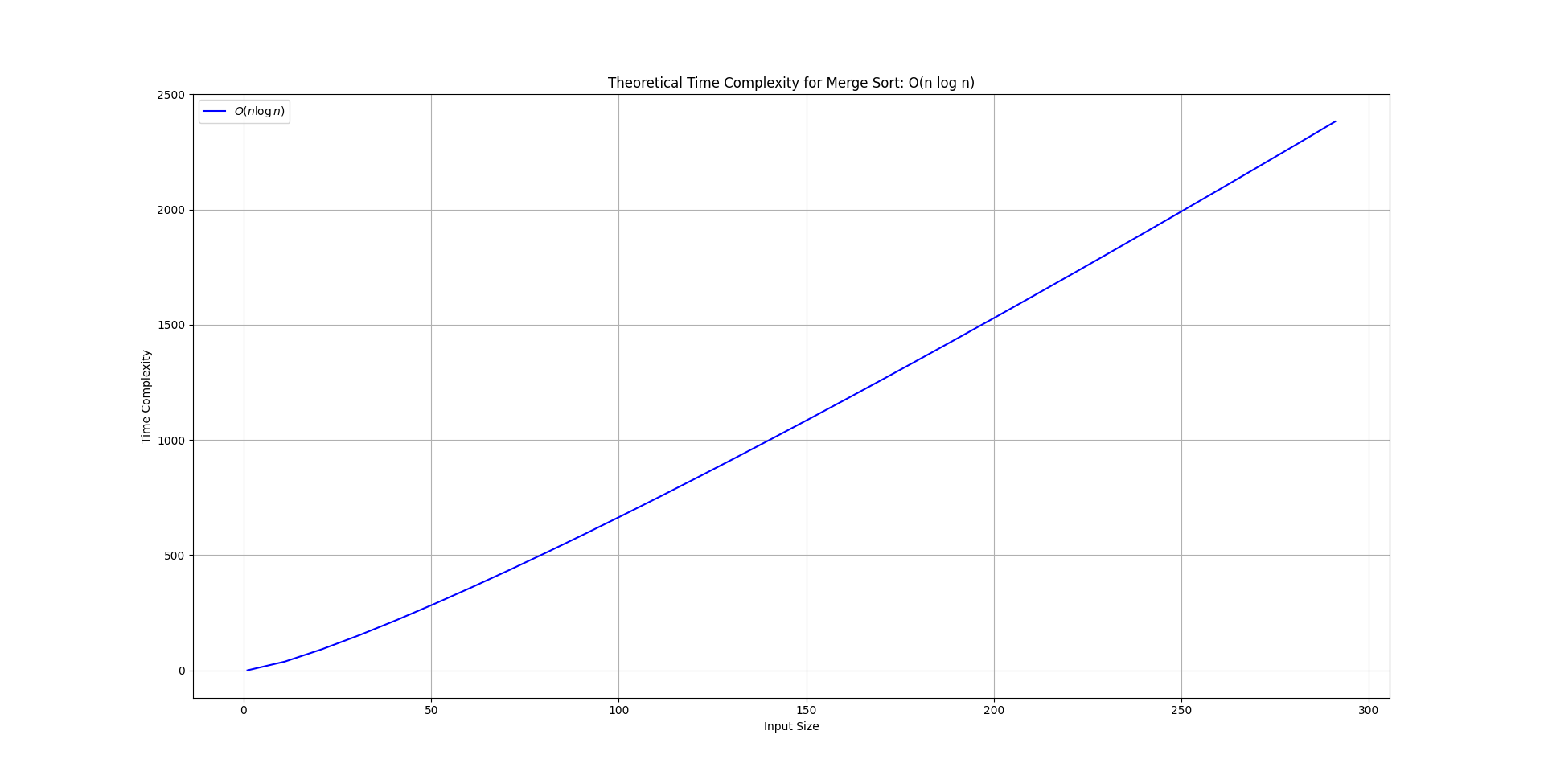
**Big O Notation Analysis:** The time complexity of Selection Sort is represented by O(n^2), where 'n' is the size of the dataset. The nested loop structure contributes to this quadratic complexity, making Selection Sort less efficient, especially as the dataset size increases. [**Figure 2**](#_toc18) displays the Big O graph for Selection Sort [4].

**Figure 2: Selection Sort Big O Notation Graph**

## b. Merge Sort:

Merge Sort uses a divide-and-conquer strategy, recursively dividing and merging sub-lists until the array is fully sorted [7].

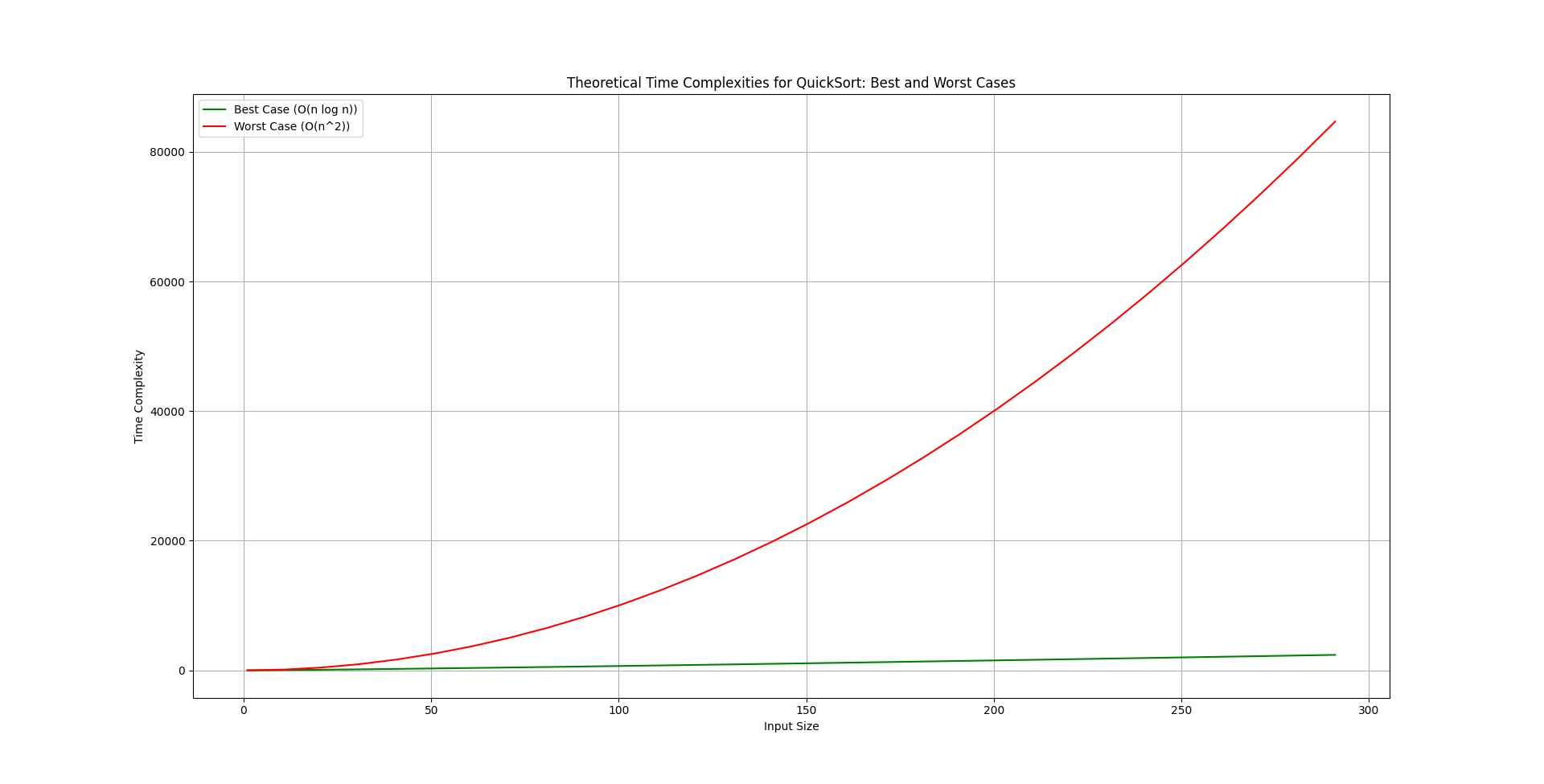
**Big O Notation Analysis:** Merge Sort maintains a stable time complexity of O(n log n) across various dataset sizes. The logarithmic time complexity ensures efficient sorting even as the dataset grows. However, the algorithm may have slightly higher constant factors due to additional memory requirements for merging sub-lists. [**Figure 3**](#_toc24)depicts the Big O graph for Merge Sort [3].

**Figure 3: Merge Sort Big O Notation Graph**

## c. Quick Sort

Quick Sort selects a 'pivot' element, divides the remaining elements based on their relationship to the pivot, and recursively sorts them [9].

**Big O Notation Analysis:** Quick Sort exhibits an average time complexity of O(n log n). In the best-case scenario, a well-chosen pivot consistently divides the array into nearly equal halves, achieving optimal efficiency. However, in the worst-case scenario, when the pivot choice leads to unbalanced partitions, the time complexity can degrade to O(n^2). [**Figure 4**](#_toc30)represents the Big O graph for Quick Sort [5].

**Figure 4: Quick Sort Big O Notation Graph**

Despite Selection Sort's simplicity, its quadratic time complexity limits practicality for extensive lists. Merge Sort, known for stability, offers consistent performance. **Based on Big O notation, Quick Sort emerges as the fastest algorithm**, with average-case efficiency and consistent out-performance, making it a preferred choice for general-purpose sorting.

# IV. Visual Performance Comparisons

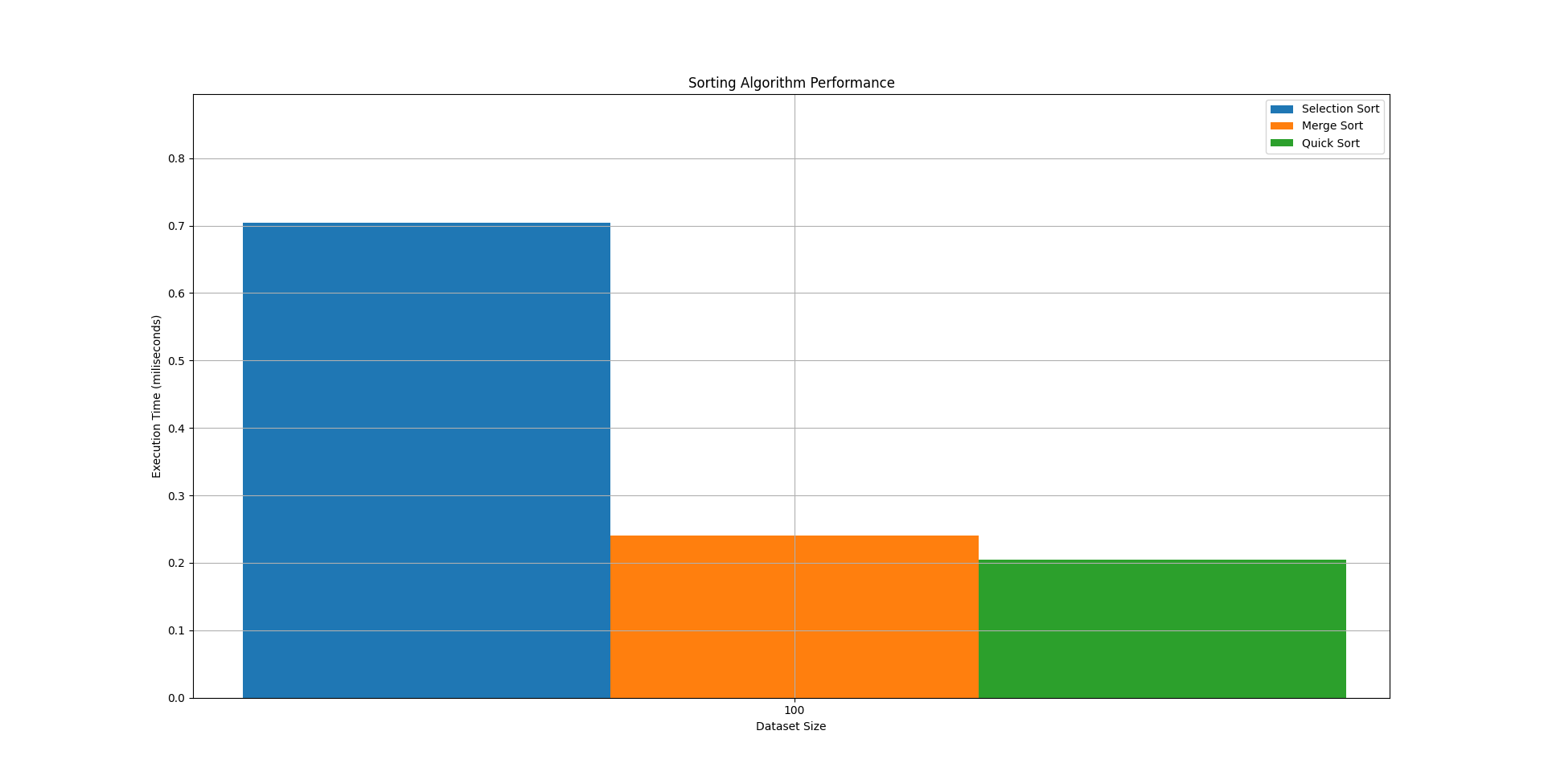
## a. Performance Comparisons

The dataset sizes (**100**, **1000**, and **10000** elements) were chosen for a comprehensive assessment of sorting algorithm performance using random 6-digit numbers. Bar charts visually represent the efficiencies of Selection Sort, Merge Sort, and Quick Sort, with the X-axis illustrating dataset sizes, and the Y-axis denoting execution time, enhancing clarity for comparing algorithmic efficiencies.

### Table 1: Array Size 100

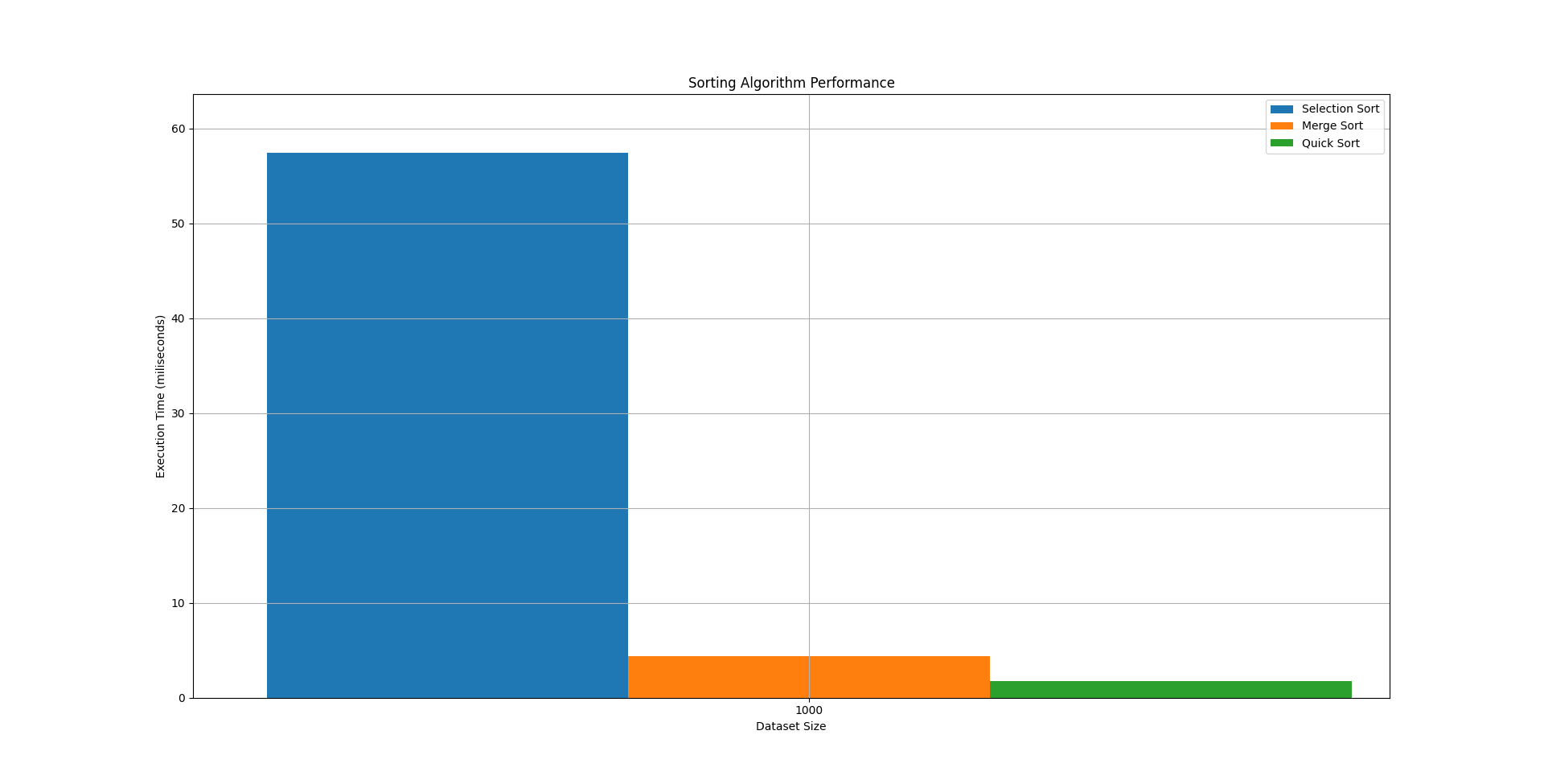
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Sorting Algorithms:** | **Execution Time 1 (milliseconds):** | **Execution Time 2 (milliseconds):** | **Execution Time 3 (milliseconds):** | **Execution Time 4 (milliseconds):** | **Execution Time 5 (milliseconds):** | **Average Execution Time (milliseconds):** |
| **Selection Sort** | **0.7003** | **0.4274** | **0.5424** | **0.5596** | **0.5496** | **0.55586** |
| **Merge Sort** | **0.2379** | **0.4239** | **0.3735** | **0.2933** | **0.3223** | **0.33058** |
| **Quick Sort** | **0.1477** | **0.1507** | **0.1635** | **0.2495** | **0.2152** | **0.18532** |

According to [**Table 1**](#_toc174), Quick Sort leads with an average execution time of 0.1477 to 0.2495ms, followed by Merge Sort at 0.2379 to 0.3735ms. Selection Sort is slower, ranging from 0.4274 to 0.7003ms. [**Figure 5**](#_toc36) confirms Quick Sort's superior efficiency over Merge and Selection Sort.

**Figure 5: Graph for 100 elements**

### Table 2: Array Size 1000

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Sorting Algorithms:** | **Execution Time 1 (milliseconds):** | **Execution Time 2 (milliseconds):** | **Execution Time 3 (milliseconds):** | **Execution Time 4 (milliseconds):** | **Execution Time 5 (milliseconds):** | **Average Execution Time (milliseconds):** |
| **Selection Sort** | **61.8589** | **46.0852** | **52.4475** | **65.848** | **52.448** | **55.13712** |
| **Merge Sort** | **3.7800** | **3.0632** | 3.9026 | **4.9804** | **2.8835** | **3.72194** |
| **Quick Sort** | **1.8488** | **2.1626** | **1.9234** | **2.1848** | **1.7885** | **1.98162** |

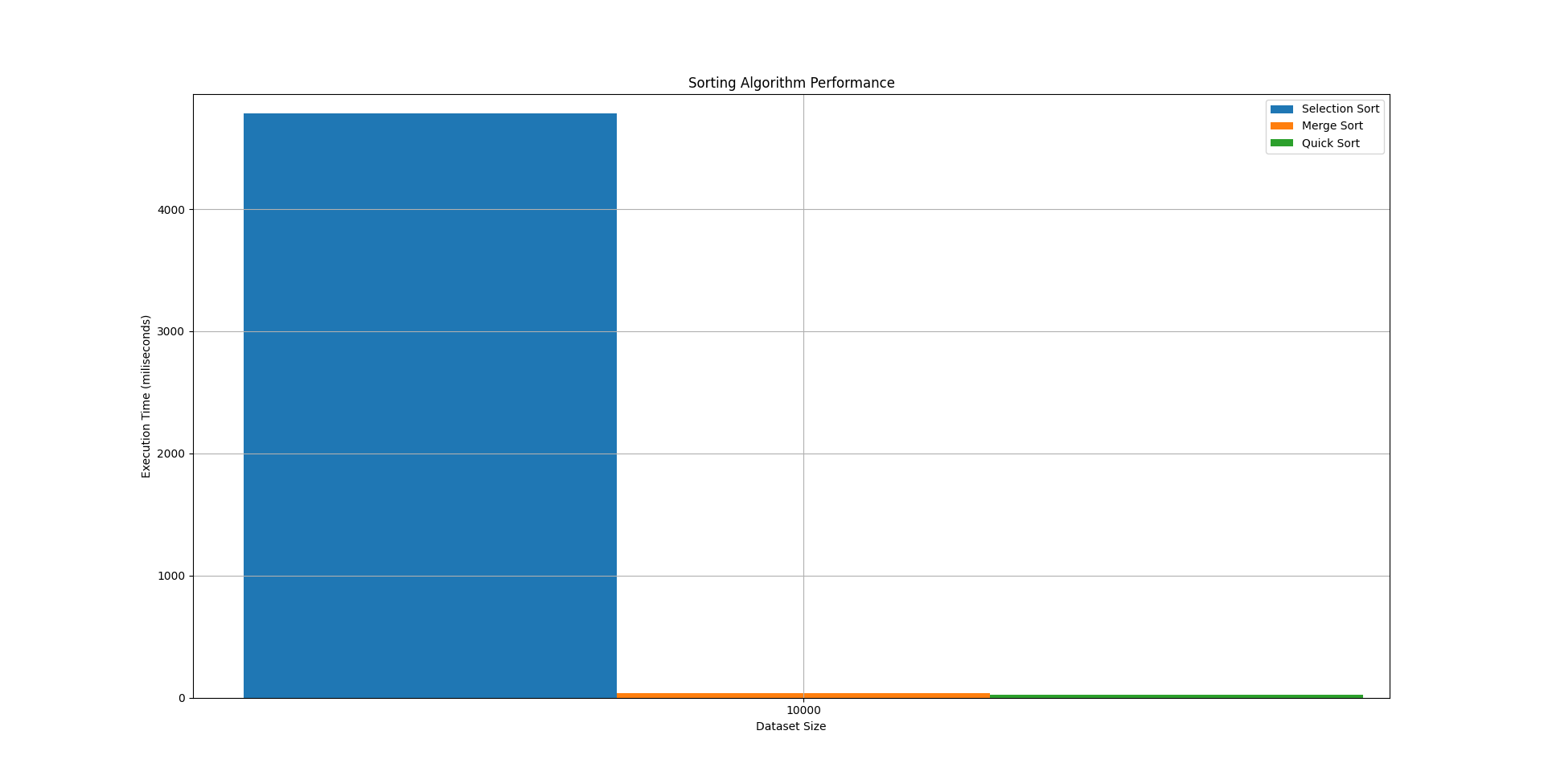
**Figure 6: Graph for 1000 elements**

In [**Table 2**](#_toc264), Quick Sort leads with an average execution time of 1.8488 to 2.1848ms, followed by Merge Sort (2.8835 to 4.9804ms). Selection Sort lags behind with execution times ranging from 46.0852 to 65.848ms. [**Figure 6**](#_toc42) confirms Quick Sort's optimal over Merge and Selection Sort.

### Table 3: Array Size 10000

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Sorting Algorithms:** | **Execution Time 1 (milliseconds):** | **Execution Time 2 (milliseconds):** | **Execution Time 3 (milliseconds):** | **Execution Time 4 (milliseconds):** | **Execution Time 5 (milliseconds):** | **Average Execution Time (milliseconds):** |
| **Selection Sort** | **4336.7777** | **4280.9905** | **4807.1223** | **4949.8158** | **4404.4324** | **4555.62774** |
| **Merge Sort** | **39.3275** | **44.5028** | **44.0276** | **41.4999** | **40.2725** | **41.92506** |
| **Quick Sort** | **25.5834** | **26.7801** | **27.7329** | **27.5196** | **26.467** | **26.81618** |

In [**Table 3**](#_toc356), Quick Sort leads with an average time of 25.5834 to 27.7329ms, while Merge Sort follows at 39.3275 to 44.5028ms. However, **Selection Sort lags significantly**, with execution times ranging from 4280.9905 to 4949.8158ms. [**Figure 7**](#_toc48)confirms Quick Sort's notable efficiency over Merge and Selection Sort.

**Figure 7: Graph for 10000 elements**

## b. Identification of the Fastest Algorithm:

Examining the data in Tables 1, 2, 3, and Figures 4, 5, 6, **Quick Sort consistently proves fastest,** owing to its optimized pivot-based partitioning. Its adaptability across varied data sizes underscores its robust speed.

In contrast, **Selection Sort consistently emerges as the slowest** due to its quadratic time complexity, especially noticeable with larger data sets. Its performance limitations make it less suitable for substantial data.

Meanwhile, **Merge Sort consistently shows balanced and reliable performance**. The divide-and-conquer approach and stable time complexity ensure dependable execution times across array sizes. Merge Sort strikes a middle ground, offering versatility in various sorting scenarios [6].

### Table 4: Summarizing algorithm comparisons across array sizes:

|  |  |  |  |
| --- | --- | --- | --- |
| **Array Size:** | **Selection Sort Comparisons** | **Merge Sort Comparisons** | **Quick Sort Comparisons** |
| **100** | **4950** | **548** | **602** |
| **1000** | **499500** | **8694** | **10458** |
| **10000** | **49995000** | **120376** | **157528** |

[**Table 4**](#_toc453) summarizes the comparisons made by these algorithms across diverse array sizes. Remarkably, Selection Sort consistently registers a higher number of comparisons compared to Merge Sort and Quick Sort, especially evident with larger arrays, highlighting its quadratic time complexity.

On the other hand, Merge Sort and Quick Sort demonstrate more efficient algorithms, maintaining lower comparison counts, notably in larger datasets. However, Quick Sort records higher comparisons than Merge Sort, attributed to its adaptive partitioning strategy. To clarify, Quick Sort's adaptive partitioning may result in a greater number of comparisons [1].

# V. Conclusion

Selecting the appropriate sorting method is pivotal. Analyzing Selection, Merge, and Quick Sort reveals their unique attributes. The efficiency of Quick Sort in balanced data highlights the importance of choosing the right method for optimal outcomes in complex computing tasks. Consequently, it is deemed the fastest method according to this report. However, Merge Sort consistently exhibits reliable performance and can outpace Quick Sort in speed as data sets grow. This emphasizes the need for a thoughtful analysis when choosing sorting algorithms for diverse computing tasks.

# VI. References

1. Al-Kharabsheh, K.S., AlTurani, I.M., AlTurani, A.M.I. and Zanoon, N.I., 2013. Review on sorting algorithms a comparative study. International Journal of Computer Science and Security (IJCSS), 7(3), pp.120-126.
2. Chivers, I. and Sleightholme, J. (2015). An Introduction to Algorithms and the Big O Notation. [online] Semantic Scholar. doi:<https://doi.org/10.1007/978-3-319-17701-4_23>.
3. Lobo, J. and Kuwelkar, S., 2020, July. Performance analysis of merge sort algorithms. In 2020 International Conference on Electronics and Sustainable Communication Systems (ICESC) (pp. 110-115). IEEE.
4. Jadoon, S., Solehria, S.F., Rehman, S.U. and Jan, H., 2011. Design and analysis of optimized selection sort algorithm. International Journal of Electric & Computer Sciences (IJECS-IJENS), 11(01), pp.16-22.
5. Xiang, W., 2011, November. Analysis of the time complexity of quick sort algorithm. In 2011 international conference on information management, innovation management and industrial engineering (Vol. 1, pp. 408-410). Ieee.
6. Pandey, R.C., 2008. Study and Comparison of various sorting algorithms. Computer Science and Engineering.
7. Marcellino, M., Pratama, D.W., Suntiarko, S.S. and Margi, K., 2021, October. Comparative of advanced sorting algorithms (quick sort, heap sort, merge sort, intro sort, radix sort) based on time and memory usage. In 2021 1st International Conference on Computer Science and Artificial Intelligence (ICCSAI) (Vol. 1, pp. 154-160). IEEE.
8. Rubinstein-Salzedo, S., 2018. Big o notation and algorithm efficiency. In Cryptography (pp. 75-83). Cham: Springer International Publishing.
9. Yang, Y., Yu, P. and Gan, Y., 2011, July. Experimental study on the five sort algorithms. In 2011 Second International Conference on Mechanic Automation and Control Engineering (pp. 1314-1317). IEEE.
10. Naz, A., Nawaz, H., Maitlo, A. and Hassan, S.M., 2021. Implementation of Selection Sort Algorithm in Various Programming Languages. International Journal, 10(3).
11. “Analysis of Algorithms | Big-O analysis - GeeksforGeeks.” Accessed: Jan. 21, 2024. [Online]. Available: https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/

# VII. Bibliographies

* Danziger, P., 2010. Big o notation. Source internet: http://www. scs. ryerson. ca/mth110/Handouts/PD/bigO. pdf, Retrieve: April, 1(1), p.6.
* Cole, R., 1988. Parallel merge sort. SIAM Journal on Computing, 17(4), pp.770-785.
* Oliveira, F.D.S. and Barbosa, V.C., 2015. Counting independent terms in big-oh notation. arXiv preprint arXiv:1512.09363.
* Darlington, J., 1978. A synthesis of several sorting algorithms. Acta Informatica, 11, pp.1-30.
* Aung, H.H., 2019. Analysis and comparative of sorting algorithms. International Journal of Trend in Scientific Research and Development (IJTSRD), 3(5), pp.1049-1053.