Algorithm Assignment1 Report

Department: RVIS, ID: 41127B041, Name: Liu, Shang-En

1 INTRODUCTION

Sorting algorithms play a fundamental role in computer science, forming the backbone of numerous applications ranging from databases to real-time systems. Among the most widely used sorting techniques are **MergeSort** and **QuickSort**, both of which utilize the divide-and-conquer paradigm. Despite their shared strategy, these algorithms exhibit distinct behaviors under different input conditions due to their structural differences in recursion, partitioning, and data handling.

This report presents a comprehensive performance analysis of MergeSort and QuickSort across a variety of input scenarios. We investigate how factors such as **input size**, **input order**, and **element duplication** influence execution time. The goal is not only to understand the asymptotic behavior of each algorithm but also to highlight practical considerations in implementation, such as **recursion thresholds** and **non-recursive optimizations**.

By systematically evaluating and visualizing the experimental results, we aim to provide insight into the practical trade-offs of using different sorting strategies, as well as the real-world implications of algorithm selection under diverse data conditions.

2 IMPLEMENTATION DETAILS

The project implements two classic divide-and-conquer sorting algorithms: **MergeSort** and **QuickSort**. Both algorithms were implemented in C++ with careful consideration for modularity, testing, and performance measurement.

For **MergeSort**, the implementation strictly follows the textbook top-down recursive structure. The array is divided recursively until subarrays of size one are reached, and then merged in a sorted manner. A non-recursive version using **insertion sort** was applied for subarrays below a certain threshold (e.g., size ≤ 32), which helps improve performance by reducing overhead caused by deep recursion.

For **QuickSort**, the implementation uses the first element as the pivot. While this approach is

simple, it deliberately exposes QuickSort to worst-case behavior in sorted or reversed inputs, which serves the purpose of this study. Like MergeSort, QuickSort also switches to **insertion sort** for small subarrays. This hybrid approach reflects common real-world optimizations found in standard libraries such as C++ STL or Python's Timsort.

To ensure consistency and reproducibility, both sorting algorithms were wrapped in functions that could measure execution time and log detailed results for further analysis.

3 EXPERIMENTAL DESIGN

The goal of the experiment is to systematically evaluate the performance of MergeSort and QuickSort under varying **input sizes** and **input types**. The experiment considers three core variables:

1. **Algorithm**: MergeSort and QuickSort.
2. **Input Size**: Arrays of size 1,000 to 10,000 in increments of 1,000.
3. **Input Type**:
    - **Random**: Elements are uniformly randomly generated.
    - **Sorted**: Elements are in increasing order.
    - **Reversed**: Elements are in decreasing order.
    - **Duplicated**: A limited set of unique elements repeated many times.

Each combination of these variables was tested multiple times (e.g., 5 runs) to ensure stability in measurements. The average execution time was recorded and exported to a .csv file for post-processing using Python.

The experiment was run on a consistent hardware and software setup, using GCC 8.1.0 (MinGW) and CMake to manage the build system. Timing was conducted using the C++ <chrono> library to ensure high-precision measurement of execution durations in milliseconds.

This design allows us to isolate the effect of each variable (e.g., input type) and observe how the algorithms respond under controlled and reproducible conditions.

4 RESULT & DISCUSSION

The experimental results reveal significant differences in the performance of MergeSort and QuickSort across various input data types and input sizes. The collected data is visualized in the chart (Figure 1), which illustrates execution time against input size, categorized by sorting algorithm and input characteristics.

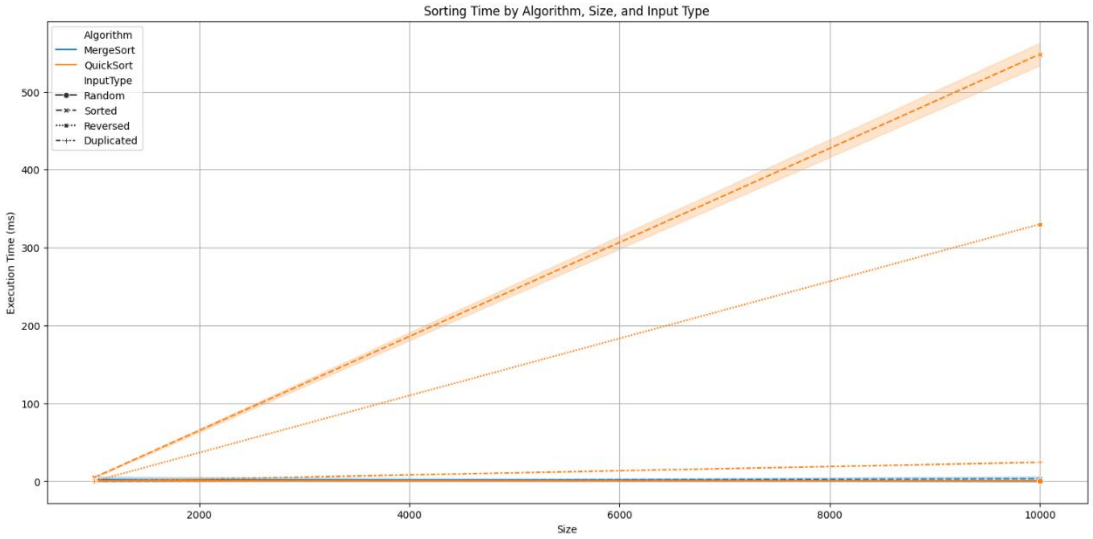

FIGURE 1. SORTING TIME COMPARISON BETWEEN MERGESORT AND QUICKSORT ACROSS INPUT TYPES AND SIZES.

4.1 MERGESORT: STABILITY ACROSS ALL INPUT TYPES

MergeSort exhibits highly consistent performance regardless of input type. Whether the array is randomly generated, already sorted, reversed, or contains many duplicated values, MergeSort maintains a nearly linear increase in execution time as input size grows. This confirms the algorithm's stability and its well-known time complexity of *O(n log n)* in all cases, due to its divide-and-conquer approach and lack of dependence on the initial data ordering.

4.2 QUICKSORT: HIGHLY SENSITIVE TO INPUT CHARACTERISTICS

In stark contrast, QuickSort's performance is strongly affected by the input data's initial order. Specifically, QuickSort demonstrates extreme inefficiency when sorting sorted or reversed arrays, where execution time skyrockets exponentially. This aligns with theoretical expectations—without a randomized or median-of-three pivot strategy, QuickSort's time complexity can degrade to $O(n^2)$ in such worst-case scenarios.

For random or duplicated inputs, however, QuickSort performs reasonably well, sometimes even outperforming MergeSort at smaller sizes. This performance gap, however, diminishes as input size increases, and MergeSort eventually becomes more efficient due to its consistent growth rate.

4.3 Implications of Input Types on Algorithm Choice

The results strongly suggest that **input type plays a critical role** in determining which sorting algorithm is preferable:

- For unknown or unpredictable data distributions, MergeSort is a safer and more reliable choice.
- For truly random or highly duplicated inputs, QuickSort can offer competitive performance, though only when enhanced with randomized pivot selection or hybrid techniques.
- Sorted or reversed arrays should be avoided with naive QuickSort implementations, or mitigated by using advanced pivot strategies.

4.4 Scalability and Performance Trends

Both algorithms display time growth proportional to input size, though with differing slopes. MergeSort's growth is consistently shallow, while QuickSort shows explosive time increases under certain input patterns. These trends confirm theoretical expectations and validate the design of our experimental setup.

5 Conclusion

The results of this experiment reaffirm the theoretical strengths and limitations of MergeSort and QuickSort. MergeSort exhibits consistently stable performance regardless of input structure, due to its predictable divide-and-merge strategy. In contrast, QuickSort’s execution time is highly sensitive to input order, performing well on random data but suffering on sorted and reversed inputs, especially when using a naive pivot selection strategy.

Our findings also confirm the practical benefit of hybridizing recursive sorts with **insertion sort** for small subarrays, improving performance by reducing recursion overhead. Additionally, input duplication shows that MergeSort maintains efficiency, while QuickSort's

performance can degrade due to unbalanced partitioning.

These insights are crucial for developers and algorithm engineers who must choose appropriate sorting strategies depending on data characteristics. Future extensions of this work could include evaluating more robust pivot strategies (e.g., median-of-three), comparing with other sorting algorithms such as Timsort or HeapSort, and profiling memory usage alongside execution time to provide a more holistic performance assessment.