**Sorting Algorithm Comparator V1**

Dwight Abrahams

CSU Global

CSC506-1: Principles of Software Development

Professor Pubali Banerjee

March 9, 2025

**Sorting Algorithm Comparator V1**

# ****Purpose of the Program****

This paper details the planning, implementation, testing, and analysis of the Sorting Algorithm Comparator project. The project aims to implement core sorting algorithms, namely Bubble Sort, Merge Sort, and Quick Sort—and compare their computational efficiencies. The project encompasses the development of a codebase, empirical testing on random datasets, and visualization of performance metrics. The analysis presented herein focuses on observed execution times, encountered obstacles, and the technical and project management skills acquired during the process.

### I. Algorithm Implementation and Testing

#### A. Code Development

A primary Python file, sorting\_comparator.py, was created using Python 3.x with standard libraries (time, random, argparse, copy) and matplotlib (via pip). Git tracked version control. Core algorithms include Bubble, Merge, and Quick Sort; placeholder functions for Insertion and Heap Sort are provided. Each algorithm is documented with inline comments and uses deep copies for fair comparison. A random list generator and argparse allow input customization. Performance is measured using time.perf\_counter, with results stored in a dictionary. The main block executes algorithms, logging execution times. (e.g., Bubble: 0.047721 sec, Merge: 0.001353 sec, Quick: 0.001273 sec) and visual results in a bar chart.

### II. Results and Discussion

The empirical analysis of the sorting algorithms reveals significant performance differences. As shown in the recorded execution times, both Merge Sort (0.001353 seconds) and Quick Sort (0.001273 seconds) vastly outperform Bubble Sort (0.047721 seconds). This substantial discrepancy aligns with theoretical expectations, given the inherent computational complexities of these algorithms.

Bubble Sort’s slower performance is attributable to its O(n2)O(n^2)O(n2) time complexity, which becomes prohibitive as input size increases. In contrast, Merge Sort and Quick Sort typically operate in O(nlog⁡n)O(n \log n)O(nlogn) time under average conditions. The near-identical performance of Merge Sort and Quick Sort in these tests suggests that for the given dataset size, both algorithms are highly efficient. The slight advantage observed in Quick Sort could be due to lower constant factors in its implementation for the tested cases.

The bar graph (Figure 2 in Appendix A) provides a visual representation of these findings. The dramatic height difference between Bubble Sort’s bar and those of Merge Sort and Quick Sort visually underscores the performance gap. This visualization not only supports the empirical data but also serves as a compelling tool for further analysis and presentation.

Furthermore, the iterative testing phase highlighted several challenges:

* **Debugging and Consistency:** Ensuring that each algorithm processed an identical dataset required careful management of data copies.
* **Measurement Precision:** Variations in system load occasionally affected execution time measurements, necessitating multiple tests for accuracy.

The project facilitated the development of robust programming practices, enhanced debugging skills, and improved familiarity with performance measurement techniques. Additionally, integrating visualization into the workflow provided valuable experience in presenting data effectively.

### III. Conclusion

The Sorting Algorithm Comparator project successfully demonstrated the comparative efficiencies of core sorting algorithms. Through meticulous implementation, testing, and visualization, the project confirmed that algorithms with lower theoretical time complexities (Merge Sort and Quick Sort) outperform less efficient methods (Bubble Sort) in practical scenarios.

The systematic approach to algorithm implementation, combined with the use of command-line parameter customization and rigorous performance measurement, ensured a thorough evaluation of each method. The challenges encountered during the testing process provided learning opportunities that enhanced both technical and project management skills.

Overall, this project not only reinforces academic and technical competencies but also underscores the importance of structured planning and empirical validation in software development.

# References

# Appendix A (photos)

A black background with white text

Description automatically generated

Figure 1: Console output

A graph with blue bars

Description automatically generated with medium confidence

Figure 2: Mathplotlib graph