**Sorting Algorithm Comparator V2**

Dwight Abrahams

CSU Global

CSC506-1: Principles of Software Development

Professor Pubali Banerjee

March 23, 2025

**Sorting Algorithm Comparator V2**

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

This paper details the planning, implementation, testing, and analysis of the Sorting Algorithm Comparator project. The primary objective was to implement five sorting algorithms (Bubble Sort, Merge Sort, Quick Sort, Insertion Sort, and Heap Sort), measure their performance across 20 runs, and analyze the empirical data to determine time complexity trends. The study highlights the obstacles encountered during development, discusses optimization techniques employed, and reflects on the skills acquired throughout the process.

### I. Algorithm Implementation and Testing

#### A. Code Development

A primary Python file, *sorting\_comparator\_v1.py*, was developed using Python 3.12 with standard libraries (time, random, argparse, copy) and the external library matplotlib (installed via pip). Core algorithms—Bubble Sort, Merge Sort, and Quick Sort—were implemented with detailed inline comments and docstrings. Insertion Sort and Heap Sort were fully implemented using standard algorithmic approaches (insertion into a sorted subarray and in-place heap construction with heapify, respectively). The input dataset is generated using a customizable function (via argparse), ensuring consistent test conditions through deep copies of the dataset. Each algorithm’s execution time was measured 20 times using time.perf\_counter(), and average times and standard deviations were calculated. The results were tabulated and visualized as a bar chart (Appendix A) for clear comparison.

### II. Results and Discussion

The empirical evaluation produced the following average execution times (over 20 runs):

* **Bubble Sort:** 0.046466 sec (Std Dev: 0.002645)
* **Merge Sort:** 0.001628 sec (Std Dev: 0.000309)
* **Quick Sort:** 0.001427 sec (Std Dev: 0.000197)
* **Insertion Sort:** 0.018456 sec (Std Dev: 0.000955)
* **Heap Sort:** 0.001897 sec (Std Dev: 0.000247)

These results indicate that Bubble Sort, with its O(n²) time complexity, is significantly slower than the other algorithms, which generally perform in O(n log n) time on average. The analysis confirms that optimizations such as early termination in Bubble Sort and an efficient in-place heapify routine in Heap Sort positively influence performance. Future improvements could include refining Quick Sort to be fully in-place and exploring hybrid approaches for smaller subarrays.

The data underscores the theoretical underpinnings of sorting algorithm complexity. Bubble Sort’s performance deteriorates rapidly with increasing input size, while Merge Sort and Quick Sort maintain consistent performance due to their divide-and-conquer strategies. The implemented optimizations—early termination in Bubble Sort and effective heap construction in Heap Sort—demonstrate the practical benefits of algorithmic refinements. Additionally, the rigorous measurement process and repeated runs ensured reliable data collection. The project fostered enhanced debugging techniques, performance measurement proficiency, and effective data visualization skills.

### III. Conclusion

The Sorting Algorithm Comparator project successfully validated the expected performance differences among common sorting algorithms. Through systematic testing and optimization, the project confirmed that algorithms with lower theoretical time complexities (Merge Sort, Quick Sort, and Heap Sort) deliver superior performance compared to Bubble Sort. The experience enriched technical competencies and project management skills, emphasizing the critical role of empirical validation in software development.

# References

Bentley, J. L. (1986). Programming Pearls. *Communications of the ACM, 29*(3), 109–117.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.

# Appendix A (photos)

A screenshot of a computer

Description automatically generated

Figure 1: Console output

A graph with blue squares

Description automatically generated with medium confidence

Figure 2: Mathplotlib graph