MUBBLE Sort: A Novel Hybrid Sorting Algorithm

Yuying Li
Becky Lin
Shriraj Vaidya
Satya Ashok Dowluri

University of Kansas

EECS 700: Algorithms for HPC
Research Project Report (Group 2)

1. Abstract

Sorting is a fundamental operation in computer science with widespread applications in data processing, search optimization, and algorithm design. While classical algorithms like Merge Sort and Quick Sort offer efficient worst-case and average-case performance, their overhead on partially sorted datasets can be non-negligible. In this project, we propose a novel hybrid sorting algorithm, *Mubble Sort*, that combines the low-overhead simplicity of Bubble Sort with the scalability and stability of Merge Sort. The algorithm applies Bubble Sort to small, dynamically sliced segments of the input array, exploiting its early-stopping behavior for nearly sorted regions, and then recursively merges these segments using Merge Sort. To evaluate its effectiveness, we also perform a we compare the Mubble sort algorithm against well-established sorting methods across datasets of varying types, sizes, and orderings. Our results demonstrate that the proposed algorithm performs competitively—often outperforming traditional algorithms—especially on inputs with partially sorted and sorted arrays.

2. Introduction

Sorting is one of the most essential operations in computer science, forming the basis of efficient searching, data organization, and algorithm design [5, 7]. It is a key component in databases, information retrieval, file system management, compiler optimization, numerical simulations, and virtually every area that deals with structured data. Due to its prevalence, considerable effort has been invested in developing sorting algorithms that are not only theoretically optimal in terms of time complexity [8, 6] but also practically efficient in real-world applications [9].

Popular sorting algorithms such as Quick Sort, Merge Sort, and Heap Sort offer $\mathcal{O}(n \log n)$ average or worst-case performance and are widely used in both academic and industrial application. Although, their actual runtime is influenced not just by asymptotic bounds but also by *constant factors* — the hidden costs associated with comparisons, memory access, and data storage. For instance, Quick Sort can degrade to $\mathcal{O}(n^2)$ in worst-case, by choosing either a largest or smallest element as the pivot variable, merge sort uses additional $\mathcal{O}(n)$ memory and Heap Sort's performance is negatively impacted by poor cache locality. These *constant factors* become the most dominant cost on small or nearly sorted data. This runtime behaviour makes quadratic-time algorithms like Bubble Sort, Insertion Sort and Selection

Sort perform faster in practice in such situations. Their characteristics such as tight inner loops, minimal overhead and better cache uses make them preferable candidates for handling shorter sequences, especially as part of a hybrid sorting strategy.

Hybrid sorting algorithms are techniques that combine two or more sorting methods to take advantage of each one's strengths while minimizing their weaknesses along with optimization techniques. They typically apply simpler algorithms on small or nearly sorted segments to reduce execution-time overhead, while using more scalable algorithms for larger or more complex parts. Timsort [9], Introsort[8], and Dual-Pivot Quick Sort [6] are widely used in industry due to their strong practical performance. These hybrid algorithms combine multiple sorting strategies to handle diverse datasets efficiently and reliably.

Real-world data typically exhibits non-uniform distributions, featuring patterns like partial orderings, sorted runs and small repetitions. Common examples include the ordered nature of log file entries, partially sorted spreadsheet data, and time-sorted data chunks from various origins. This non-random nature of real-world data motivates the exploration of hybrid sorting approaches that can dynamically adjust their behavior to exploit the underlying structure of the data.

The key contributions of this work are summarized as follows:

- We design and present a new hybrid sorting algorithm called **Mubble Sort** that integrates the efficiency of **Merge Sort** with the simplicity and low-overhead benefits of **Bubble Sort**, aiming to optimize performance on diverse real-world datasets using **dynamic slicing**, **threshold tuning** and **array sortedness check**.

- We perform a detailed **empirical evaluation**, benchmarking the proposed algorithm against widely-used sorting methods as well as existing hybrid sorting algorithms to assess improvements in runtime behavior across various scenarios.

- We validate the algorithm on datasets comprising arrays of **different data types, sizes, and ordering patterns**, demonstrating its adaptability and consistent performance across heterogeneous input characteristics.

2.1 Common Sorting Algorithms

Merge Sort employs a divide-and-conquer strategy, recursively splitting the array into smaller subarrays, sorting them, and merging them back into a sorted whole. Bubble Sort, in contrast, is a straightforward comparison-based method that repeatedly swaps adjacent elements if they are in the wrong order, iterating until no more swaps are needed. Quick Sort also uses a divide-and-conquer approach, selecting a pivot element to partition the array and recursively sorting the resulting subarrays. Heap Sort transforms the array into a max-heap, repeatedly extracting the maximum element to construct the sorted array. Insertion Sort builds the sorted array incrementally, inserting each element into its correct position among the previously sorted elements. Selection Sort iteratively selects the smallest element from the unsorted portion and places it at the beginning, gradually building the sorted array.

2.2 Performance Considerations

For large datasets, algorithms like Quick Sort, Merge Sort, and Heap Sort are preferred due to their $\mathcal{O}(n \log n)$ time complexity. However, for small arrays (e.g., $n \leq 10$), simpler algorithms like Bubble Sort, Insertion Sort, or Selection Sort can outperform their logarithmic counterparts. This is because the constant factors ($C_{p,i,s}$) in their $\mathcal{O}(n^2)$ complexity are smaller than those ($C_{q,m,h}$) in $\mathcal{O}(n \log n)$ algorithms, making them more efficient for small n. Mathematically, for small n:

$$C_{p,i,s} \cdot n^2 \leq C_{q,m,h} \cdot n \log n$$

This observation is critical for hybrid sorting algorithms, which aim to leverage the strengths of both $\mathcal{O}(n^2)$ and $\mathcal{O}(n \log n)$ algorithms.

3. Literature Review

Timsort, a widely adopted hybrid sorting algorithm used in Python and Java, combines Insertion Sort and Merge Sort to achieve robust performance on real-world data [3]. Auger et al. provide the first rigorous proof of Timsort's worst-case time complexity of $\mathcal{O}(n \log n)$, while also demonstrating its adaptability to naturally ordered sequences [3]. They show that Timsort achieves near-linear time complexity of $\mathcal{O}(n \log n + \rho)$, where ρ represents the number of pre-existing sorted runs in the input. This adaptability makes Timsort particularly effective for partially ordered datasets, setting a benchmark for hybrid sorting algorithms.

Introsort [8], combines Quick Sort, Heap Sort, and Insertion Sort to achieve robust performance. It begins with Quick Sort but switches to Heap Sort if the recursion depth exceeds $2\lfloor \log n \rfloor$, ensuring a worst-case time complexity of $\mathcal{O}(n \log n)$. For small partitions (typically $n \leq 16$), it uses Insertion Sort due to its low constant factors. Operating in-place with $\mathcal{O}(\log n)$ space for the recursion stack, Introsort avoids Quick Sort's $\mathcal{O}(n^2)$ worst-case while retaining its average-case efficiency. It is widely used in C++'s `std::sort` and `std::stable_sort` implementations.

Hybrid sorting algorithms have demonstrated superior efficiency compared to traditional sorting methods by combining the strengths of multiple techniques to optimize performance across diverse datasets. A notable example is OptiFlexSort, proposed by Abuba et al., which builds on QuickSort's framework but incorporates enhanced pivot selection using the median-of-three method and adaptive partitioning techniques inspired by other sorting algorithms [1]. Their experiments show that OptiFlexSort achieves a 10-15% improvement in execution time over traditional algorithms like Merge Sort and Heap Sort when processing large datasets, highlighting the potential of hybrid approaches to improve scalability in data-intensive applications.

Another hybrid approach, explored by Alqattan et al., involves dividing an array into two parts, applying Insertion Sort to one half and Merge Sort to the other [2]. Their comparative analysis of execution times reveals that this hybrid sort outperforms Insertion Sort in worst-case scenarios due to Merge Sort's efficiency on larger segments. However, it fails to surpass the performance of pure Merge Sort, particularly for larger arrays, suggesting that simplistic hybrid strategies may struggle to balance the trade-offs between different sorting methods on varying input sizes.

In the context of resource-constrained environments, Ayodele et al. introduced the KSU Hybrid Sort, which integrates Quick Sort and Insertion Sort with a cut-off size threshold [4]. This algorithm leverages Quick Sort's speed for large subarrays and Insertion Sort's efficiency for smaller ones, achieving significant energy savings—10.5 joules compared to 61.5 joules for Quick Sort at 100,000 elements in Python. These results underscore the importance of tailoring hybrid algorithms to specific hardware constraints, particularly for energy-efficient computing on devices with limited resources.

4. Methodology

4.1 Algorithm Design: Mubble

In the development of Mubble, we examined the strengths of Bubble Sort and Merge Sort. Merge Sort is known for its divide-and-conquer approach, recursively dividing arrays down to individual elements before merging. To optimize this process, we introduced a new idea. Rather than dividing the array all the way into single elements, we will divide it into smaller subarrays based on a predefined threshold. Then, given that Bubble Sort performs well on small arrays, we will use it to sort these smaller subarrays. After sorting, the subarrays will then be merged using Merge Sort. This approach combines the simplicity of Bubble Sort on small-sized arrays with the scalability of Merge Sort. Figure 1 shows the process of Mubble sort.

4.2 Algorithm Enhancements

To further optimize Mubble's performance, we implemented several enhancements beyond the basic hybrid structure:

- **Threshold Tuning:** Threshold size plays a critical role in the execution times of our algorithm. We experimented with threshold values ranging from 8 to 64. Since Bubble Sort's performance gets worse on larger arrays, it was determined that the best threshold size is 10.
- **Dynamic Slicing:** Dynamic slicing is used when dividing the array into smaller subarrays. The algorithm checks whether a subarray is in sorted order. If it is, the algorithm continues appending elements to the subarray until the next element is no longer in sorted order. This way, naturally sorted subarrays can be left as-is, and Bubble Sort is skipped for them.
- **Sortedness Check:** Inspired by a feature in TimSort, we implemented a pre-processing step to detect if the entire input array is already in sorted order. If the array is found to be in order, no sorting is performed. The helps reduce unnecessary computation.
- **Reverse Order Check:** The reverse order check is applied after the array has been divided into subarrays. If any subarray is found to be in reverse order, it is reversed directly without additional sorting.

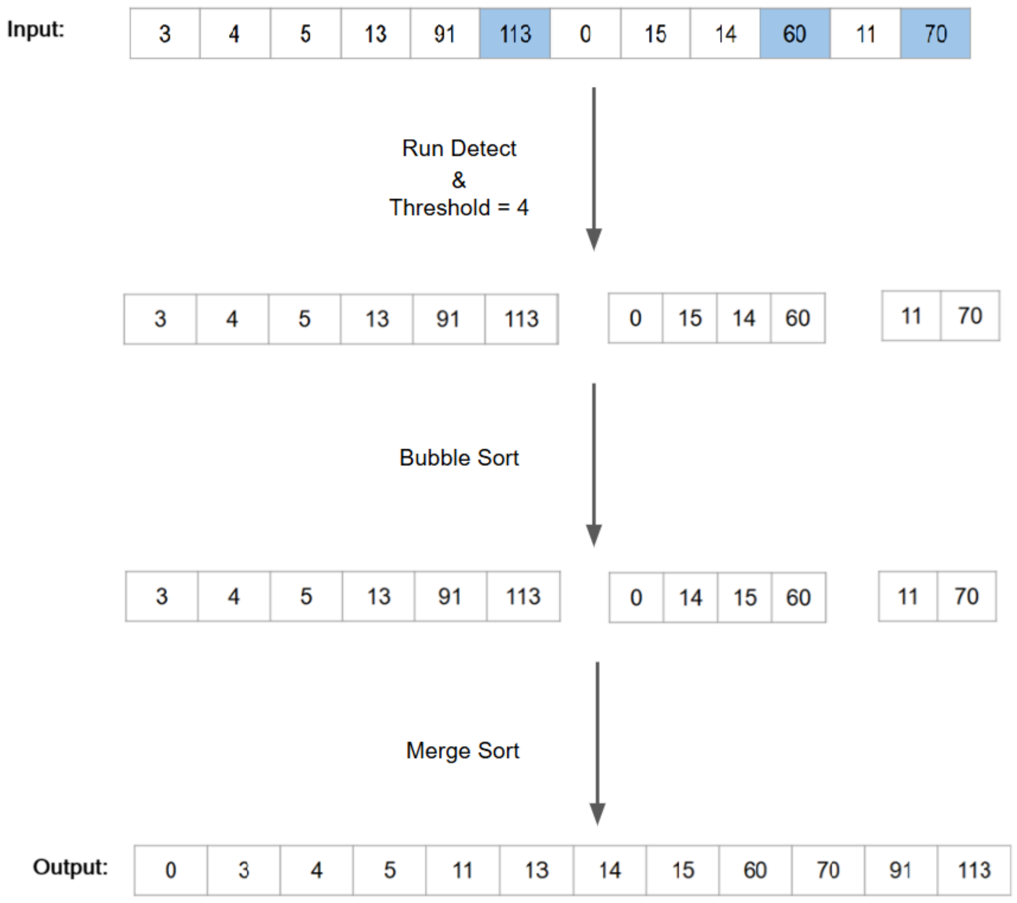

Figure 1: Shows the process of Mubble sort. Pivot highlights blue.

4.3 Experimental Evaluation

To understand how well Mubble performs in practice, we set up a series of experiments and compared it against two hybrid and two standard sorting algorithms: Timsort, Introsort, Merge Sort, and Bubble Sort. Our goal wasn't just to measure speed, but also to see how consistently each algorithm behaves across different types of input—whether the data is already sorted, completely jumbled, or somewhere in between.

4.4 Dataset Configuration

We used four types of input datasets to reflect a range of realistic scenarios. The sorted dataset represents the ideal case, where all elements are already in ascending order. The reversed dataset does the opposite—it's arranged in descending order, which is usually the worst-case scenario for algorithms like Bubble Sort. The half-sorted dataset is a mix, with about 50% of the elements in sorted order and the rest randomly shuffled. This helps us test how well the algorithm can take advantage of partial order. Finally, the random dataset contains fully random numbers, which is a typical case for general algorithm performance testing.

Each dataset type was created at five different sizes: 1,000, 5,000, 10,000, 50,000, and 100,000 elements. To make sure our results were reliable, we ran each test 100 times for every combination of dataset type, size, and sorting algorithm.

4.5 Performance Metrics and Environment

For each test run, we recorded the execution time and computed the mean, median, maximum, minimum, and standard deviation to analyze performance trends and consistency. All experiments were conducted under controlled and identical hardware conditions to ensure reproducibility. Tests were performed on an I2S computing cluster equipped with a GPU g021, a 4 GHz processor, and 50 GB of memory.

5. Results

This section presents the performance of Mubble compared with Timsort, Introsort, Merge Sort, and Bubble Sort across four input types: random, reversed, half-sorted, and sorted.

We created a bar chart for the results on each dataset; these figures show normalized execution time, where Bubble Sort is used as the baseline (set to 1.0). The height of each bar represents the mean execution time across 100 runs, and lower bars indicate better performance.

The error bars on top of the bars show the standard deviation, reflecting runtime consistency.

5.1 Random Array

In the random array benchmark, IntroSort had the fastest execution time, followed closely by TimSort and Mubble. Merge Sort was consistently the slowest across all input sizes.

At size 100,000, Mubble ran 98.9% faster than Merge Sort and 92.9% faster than Bubble Sort. Compared to TimSort, Mubble was only 0.3% slower, indicating that the two algorithms performed nearly identically. The relative ranking of the algorithms remained consistent across all dataset sizes. See Figure 2.

5.2 Reversed Array

On reversed input, Mubble showed the best overall performance across all dataset sizes.

At size 100,000, it was 94.3% faster than Merge Sort, 89.1% faster than IntroSort, and 99.9% faster than Bubble Sort.

Mubble also outperformed TimSort by 8.8%, making it the fastest among all tested algorithms in this scenario. IntroSort was the slowest on reversed input, especially as the array size increased. See Figure 3.

5.3 Half-Sorted Array

In the half-sorted array benchmark, TimSort and IntroSort outperformed Mubble across all input sizes. However, Mubble was still faster than Merge Sort.

At size 100,000, Mubble was 90.8% faster than Merge Sort and 99.9% faster than Bubble Sort. However, it was 23.1% slower than TimSort and 17.6% slower than IntroSort.

This result likely reflects Bubble Sort's weakness in handling partially ordered data, as it continues to perform many redundant comparisons and swaps. In contrast, both TimSort and IntroSort use Insertion Sort for segment-level sorting, which is more efficient for inputs that are already partially sorted. See Figure 4.

5.4 Sorted Array

In the sorted array case, the performance gap between algorithms was especially large.

To clearly visualize the results, we split the figure into two subgraphs: one comparing Merge Sort and IntroSort, and the other comparing TimSort and Mubble.

At size 100,000, Mubble was 97.6% faster than Merge Sort, 99.2% faster than IntroSort, and 100.0% faster than Bubble Sort. Its performance closely matched TimSort, with only a 0.3% difference.

These two algorithms both detect sorted segments and return them without additional processing, which greatly reduces runtime on fully ordered inputs. See Figure 5.

5.5 Limitation

Mubble is faster than Bubble Sort, Merge Sort, and even IntroSort across various array types and sizes, but still slightly slower than TimSort.

Our main limitation comes from the use of Bubble Sort for sorting short segments. While we implemented early-stopping logic to improve its efficiency, Bubble Sort remains slower than Insertion Sort. This is because Bubble Sort performs more swaps and compares elements repeatedly, even when the data is nearly sorted.

Insertion Sort, on the other hand, shifts elements with fewer operations and tends to be more efficient on partially sorted arrays.

Since segment-level sorting is called frequently in hybrid algorithms like Mubble, the choice of sorting method has a noticeable impact on overall performance. As a result, this design choice becomes a limiting factor, especially when compared to TimSort, which uses Insertion Sort for segment sorting. However, our results still support that a simple hybrid design like Mubble can achieve meaningful improvements over traditional algorithms on a variety of input types.

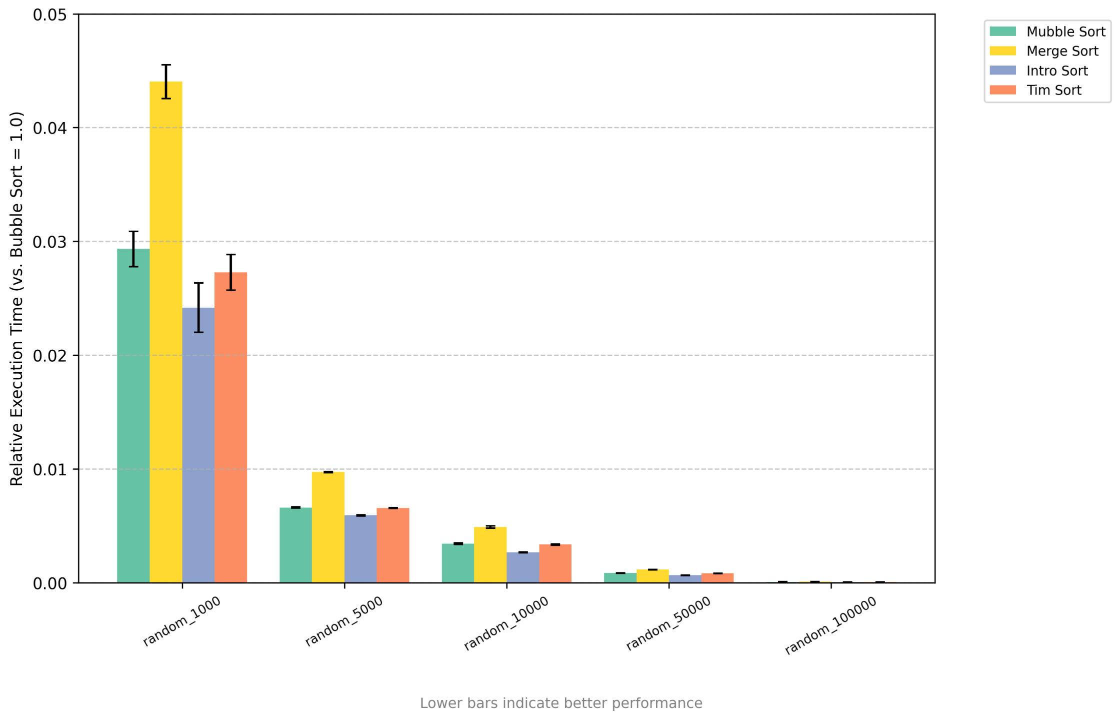

Figure 2: Relative Execution Time on Random Arrays

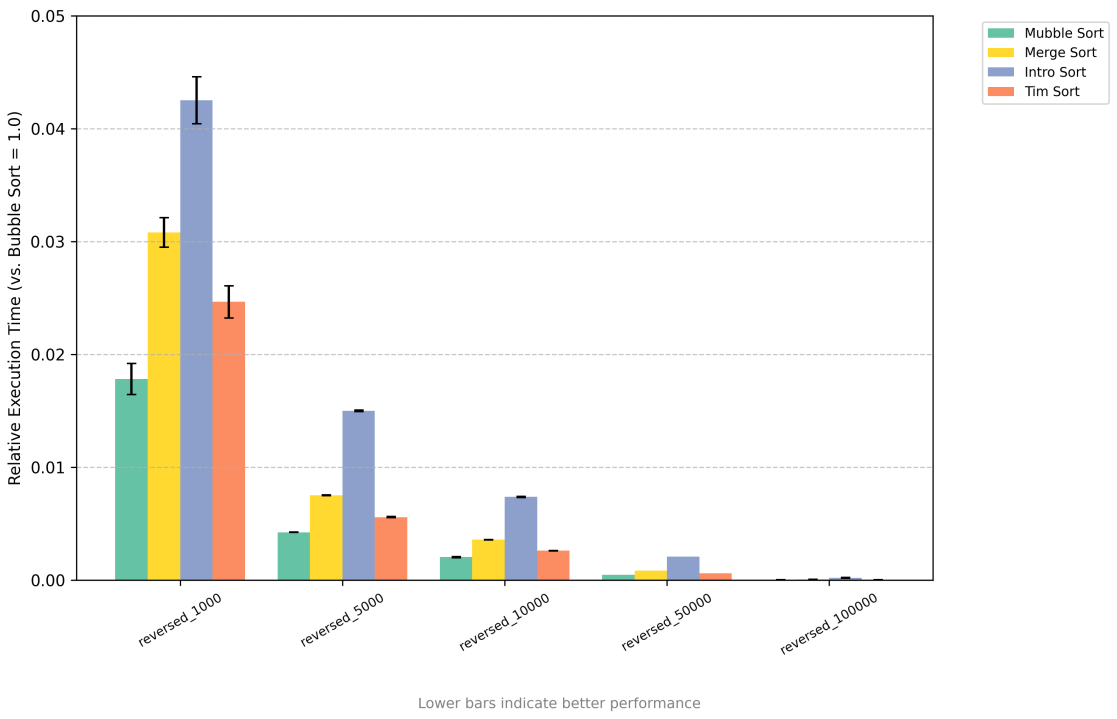

Figure 3: Relative Execution Time on Reversed Arrays

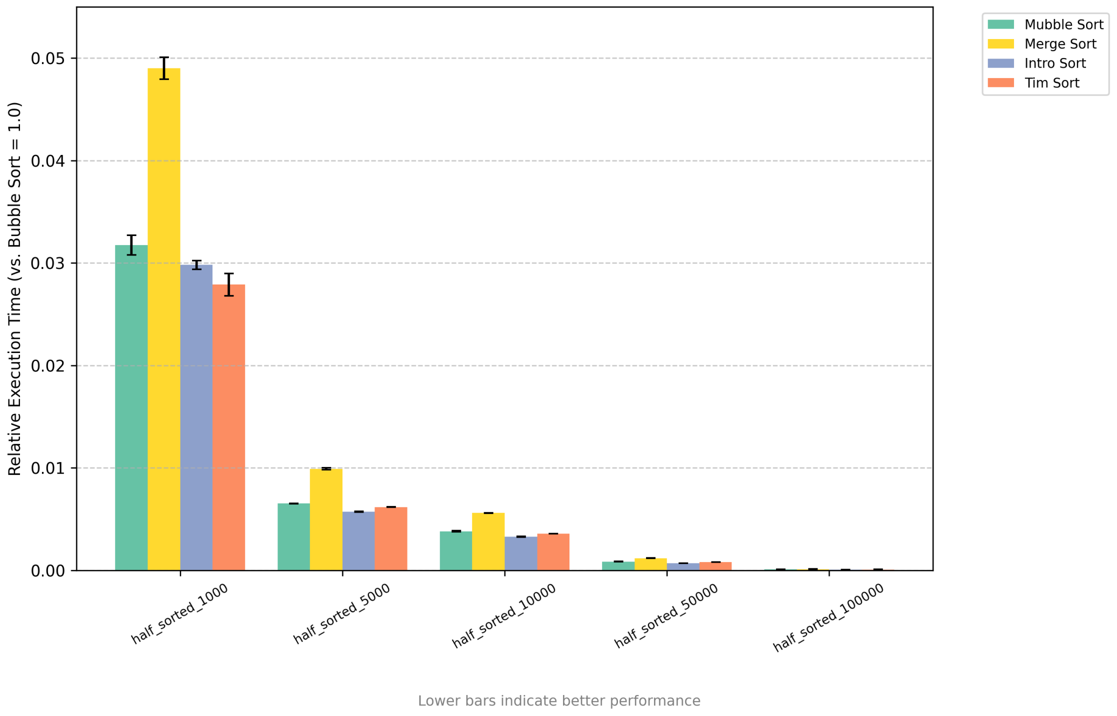

Figure 4: Relative Execution Time on Half-Sorted Arrays

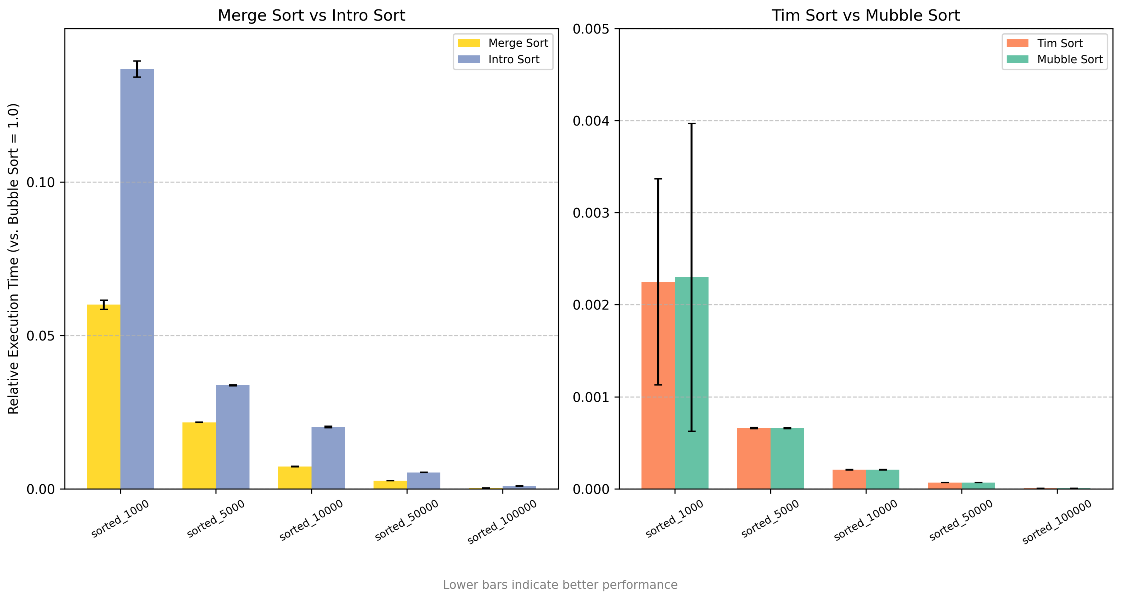

Figure 5: Relative Execution Time on Sorted Arrays

6. Conclusion

This project presents Mubble, a hybrid sorting algorithm that uses Bubble Sort for short segments and Merge Sort for merging. Through benchmark tests on random, reversed, half-sorted, and sorted arrays, we show that Mubble consistently outperforms standard algorithms like Bubble Sort, Merge Sort, and even IntroSort. In many cases, Mubble ranks just behind TimSort, a highly optimized hybrid algorithm widely used in practice.

Mubble's performance highlights the value of hybrid sorting: by combining simple components with adaptive logic, such as threshold tuning, run detection, and sortedness checks, it achieves both speed and flexibility across a range of input types. Through this design, Mubble was able to achieve both speed and stability across different input types.

Looking forward, there are several promising directions to improve Mubble. Replacing Bubble Sort with lighter alternatives like Insertion, Selection, or Shell Sort may significantly reduce overhead. We can also explore randomized or adaptive combinations of sorting strategies for both segment sorting and merging. In addition, adopting advanced techniques from existing hybrid algorithms may further improve adaptability. Benchmark comparisons with a broader range of hybrid algorithms can also offer deeper insights.

In summary, Mubble demonstrates that even a simple hybrid architecture can achieve competitive performance. It offers a solid foundation for future work on hybrid sorting strategies.

References

[1] Nelson Seidu Abuba, Edward Yellakuor Baagyere, Callistus Ireneous Nakpih, and Japheth Kodua Wiredu. Optiflexsort: A hybrid sorting algorithm for efficient large-scale data processing. 2023.

[2] Masuma M. Alqattan, Shahad B. AlQarni, Zainab R. Alramadan, and Razan A. Al Sari. Comparison of insertion, merge, and hybrid sorting algorithms using c++. 2023.

[3] Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivoteau. On the worst-case complexity of timsort. 2020.

[4] Oluwakemi Sade Ayodele, Sheidu Audu Yakubu Owoeye, and Leke Joseph Oloruntoba. Development of a novel hybrid sorting algorithm for energy efficiency in resource-constrained devices. 2023.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[6] Vasileios Iliopoulos and David B. Penman. Dual pivot quicksort. *CoRR*, abs/1503.08498, 2015.

[7] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998.

[8] David R Musser. Introspective sorting and selection algorithms. *Software—Practice & Experience*, 27(8):983–993, 1997.

[9] Tim Peters. Timsort: A stable adaptive sorting algorithm. *Python Software Foundation*, 1997. `https://svn.python.org/projects/python/trunk/Objects/listsort.txt`.