PERFORMANCE ANALYSIS OF AN ADAPTIVE SORTING ALGORITHM BASED ON INPUT CHARACTERISTICS

BY

ABHISHEK HANAMANT KINAGI

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

EMAIL: kinagiabhishek842@gmail.com

FEBRUARY ,2026

ABSTRACT:

Sorting performance largely depends on the characteristics of input data. Conventional sorting algorithms such as Insertion Sort, Merge Sort, and Quick Sort perform efficiently only under specific conditions and may degrade in other scenarios. This paper presents an adaptive sorting algorithm that dynamically selects an appropriate sorting technique based on input size, disorder ratio, and presence of duplicate elements.

The proposed method first analyses the input dataset and then chooses the most suitable sorting algorithm to optimize performance. Experimental evaluation was conducted on datasets ranging from 1,000 to 1,000,000 elements, and execution times were measured in nanoseconds. The results show that while Insertion Sort is inefficient for large inputs, Merge Sort and Quick Sort scale well. The adaptive approach introduces minimal overhead and achieves competitive performance across different input conditions, demonstrating improved robustness compared to fixed sorting strategies.

KEYWORDS:

Adaptive Sorting, Sorting Algorithms, Performance Analysis, Quick Sort, Merge Sort

INTRODUCTION:

Sorting is a fundamental operation in computer science and plays a crucial role in a wide range of applications such as database management, information retrieval, data analytics, and operating systems. Efficient sorting directly impacts the performance of higher-level algorithms including searching, indexing, and optimization tasks. As data sizes continue to grow and diversify in structure, the choice of an appropriate sorting algorithm becomes increasingly important.

Traditional sorting algorithms such as **Insertion Sort**, **Merge Sort**, and **Quick Sort** exhibit different performance characteristics depending on the nature of the input data. For example, Insertion Sort performs efficiently on nearly sorted datasets but degrades significantly for large random inputs. Merge Sort provides stable and predictable performance with a time complexity of $O(n\log\ n)$but incurs additional memory overhead. Quick Sort is generally fast in practice but can suffer from poor performance in worst-case scenarios such as already sorted or reverse-sorted inputs.

Most real-world datasets are **not purely random**; instead, they often exhibit properties such as partial ordering, repeated elements, or specific distribution patterns. However, conventional sorting implementations typically apply a single algorithm without considering these input characteristics. This leads to suboptimal performance in many practical situations.

To address this limitation, **adaptive sorting techniques** aim to analyze the input data and dynamically select the most suitable sorting strategy. By leveraging simple input metrics—such as the degree of disorder or the presence of duplicate elements—an adaptive approach can exploit favourable conditions and reduce unnecessary computational overhead.

In this paper, we propose and evaluate an **adaptive sorting framework** that performs lightweight input analysis and selects an appropriate sorting algorithm accordingly. The proposed system benchmarks traditional sorting algorithms alongside the adaptive approach and compares their performance across varying input sizes. The goal is to demonstrate that adaptive algorithm selection can achieve competitive or improved performance without significant analysis overhead.

RELATED WORK:

Traditional sorting algorithms such as Insertion Sort, Merge Sort, and Quick Sort have been extensively studied and widely used in various applications. Insertion Sort ***performs efficiently on small or nearly sorted datasets*** but becomes inefficient for large, randomly ordered ***inputs due to its quadratic time complexity***.

Merge Sort guarantees stable performance with a ***time complexity of O(n log n) regardless of input order***, but it ***requires additional memory***, making it less suitable for memory-constrained environments. Quick Sort, while ***highly efficient on average, may degrade to O(n^2) in worst-case scenarios such as already sorted or reverse-sorted*** inputs.

To overcome these limitations, ***adaptive sorting techniques*** have been proposed, where the algorithm dynamically selects the most suitable sorting strategy based on characteristics of the input data***. Existing adaptive approaches focus on factors such as input size, degree of disorder, and presence of duplicate elements***. This work builds upon these ideas by implementing a lightweight adaptive sorting framework that selects an optimal sorting algorithm at runtime using simple input analysis metrics.

METHODOLOGY:

INPUT ANALYSIS:

Before applying any sorting algorithm, the input array is analysed to extract key characteristics that influence sorting performance. The parameters considered include input size, degree of disorder, and proportion of duplicate elements.

The **disorder ratio** is computed by counting adjacent inversions in the array, which provides an estimate of how far the array is from being sorted. A lower disorder ratio indicates a nearly sorted input.

The **duplicate ratio** is calculated as the fraction of repeated elements in the array. High duplicate presence can negatively affect certain algorithms such as Quick Sort.

This lightweight analysis enables the system to make an informed decision without adding significant computational overhead.

ADAPTIVE DECISION STRATEGY:

Based on the input analysis, an adaptive decision strategy is employed to select the most suitable sorting algorithm. The decision rules are designed to balance efficiency and simplicity.

- If the input size is small or the array is nearly sorted, Insertion Sort is selected due to its low overhead and efficiency in such cases.
- If the array contains a high proportion of duplicate elements, Merge Sort is chosen to ensure stable and consistent performance.
- In all other cases, Quick Sort is applied as it provides superior average-case performance for large, randomly distributed datasets.

SORTING ALGORITHMS USED:

The adaptive framework integrates three classical sorting algorithms.

- **Insertion Sort** is used for small or nearly sorted inputs due to its minimal overhead and linear behaviour in best cases. If size <50 or disorder <0.05.
- **Merge Sort** provides guaranteed O(n log n) performance and is preferred when duplicate elements are dominant. If duplicates>0.5.
- **Quick Sort** is employed for large and randomly ordered inputs because of its excellent average-case efficiency.

SYSTEM WORKFLOW:

The overall workflow of the adaptive sorting system begins with receiving the input array. The analyser module evaluates the array characteristics and computes relevant metrics. Based on these metrics, the decision module selects the appropriate sorting algorithm. The chosen algorithm is then executed, and the sorted output is produced.

EXPERIMENTAL SETUP:

HARDWARE AND SOFTWARE ENVIRONMENT:

All experiments were conducted on a system equipped with an Intel-based processor, 16 GB RAM, running the Windows operating system.

The algorithms were implemented in **Java**, and experiments were executed using the standard **Java Development Kit (JDK)**. Execution time was measured using high-resolution time functions to ensure accurate performance comparison.

DATASET GENERATION:

Since the objective of this study is to evaluate algorithmic behaviour under controlled conditions, synthetic datasets were generated programmatically.

The datasets include:

- Randomly ordered arrays
- Nearly sorted arrays
- Reverse sorted arrays
- Arrays with high duplicate values

Array sizes of **1000, 5000, 10000, 100000 and 1000000 elements** were used to analyze scalability and performance trends.

BENCHMARKING METHODOLOGY:

Performance evaluation was conducted using a benchmarking framework that measures execution time in nanoseconds. Each sorting algorithm—Insertion Sort, Merge Sort, Quick Sort, and the proposed Adaptive Sort—was executed on identical input datasets.

To minimize noise, timing was measured using system-level time functions, and results were recorded consistently for comparison. The same dataset was reused across algorithms to ensure fairness.

EVALUATION METRICS:

The primary evaluation metric used in this study is **execution time**, measured in nanoseconds. Additionally, input characteristics such as disorder ratio and duplicate ratio were recorded to justify algorithm selection decisions made by the adaptive strategy.

RESULTS:

Results are based on Experimental data: ***Each experiment was executed three times, and the average execution time was recorded to reduce the effects of JVM warm-up and system-level noise.***

Execution time comparison of sorting algorithms for different input sizes:

Input size (N)	Insertion Sort(ns)	Merge Sort(ns)	Quick Sort(ns)	Adaptive(ns)
1000	1451433	1335033	1735300	1391100
5000	6891433	861000	300467	1067000
10000	15138267	985500	581933	2119633
100000	1086090833	9793267	6744833	19884267
1000000	103185551533	199499667	128410933	288996867

The execution times of Insertion Sort, Merge Sort, Quick Sort, and the proposed Adaptive Sorting algorithm were measured for varying input sizes. Table presents the observed execution times in nanoseconds.

From the experimental results, it is evident that Insertion Sort exhibits significantly higher execution time as input size increases, making it unsuitable for large datasets. Merge Sort and Quick Sort demonstrate better scalability, with Quick Sort consistently achieving the lowest execution time for random inputs.

The Adaptive Sorting algorithm dynamically selected an appropriate sorting technique based on input characteristics. Although it incurs a small overhead due to input analysis, its performance remains competitive and avoids worst-case behaviour across different dataset sizes.

Input size	Best Algorithm	Adaptive Time	Overhead
1000000	Quick Sort	288996867	**125.05%**

RESULT IS CALCULATED USING:

$$\textbf{Overhead (\%)} = \frac{\textbf{(Adaptive Time} - \textbf{Best Base Algorithm Time)} \times 100}{\textbf{Best Base Algorithm Time}}$$

Figure 1: Execution time comparison of sorting algorithms across varying input sizes (log–log scale). Results are averaged over multiple runs.

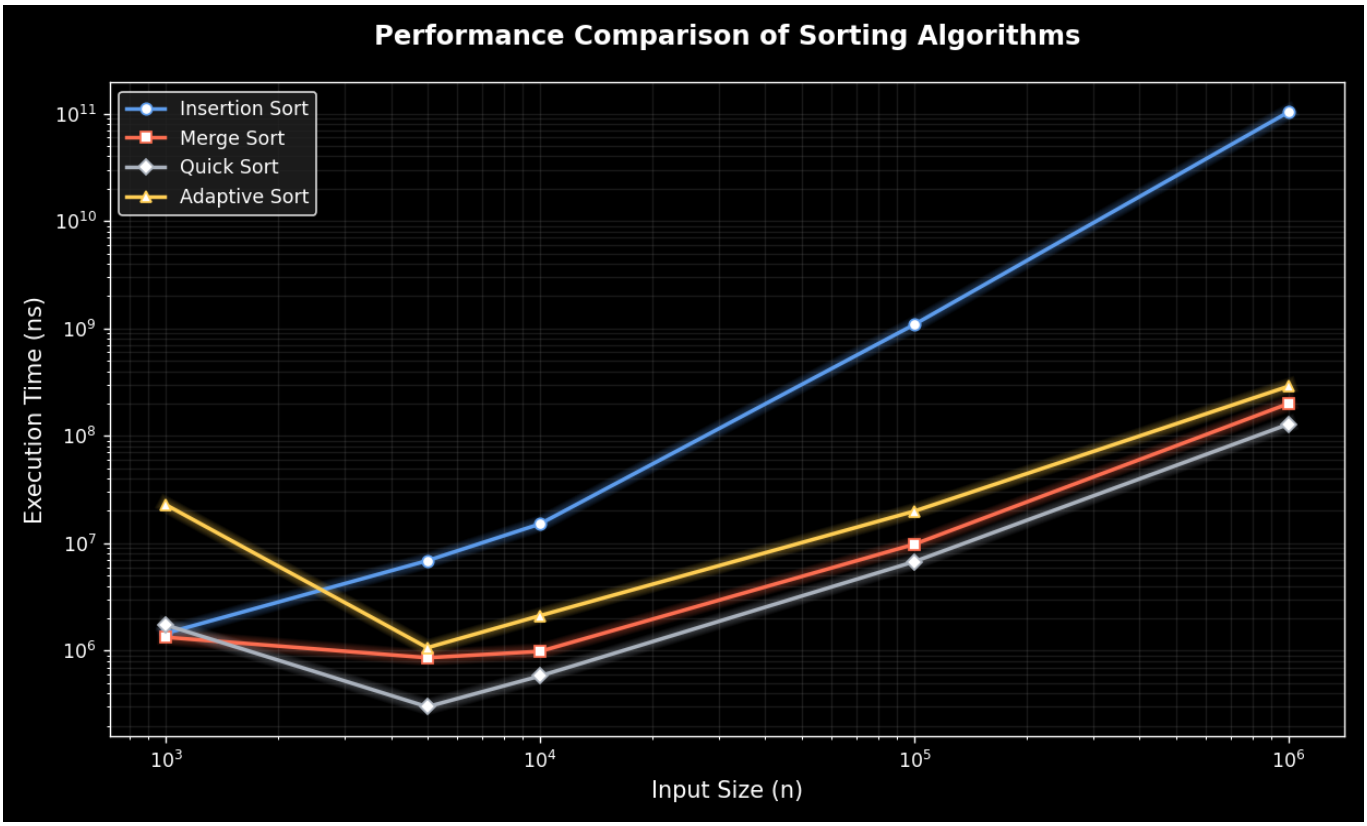
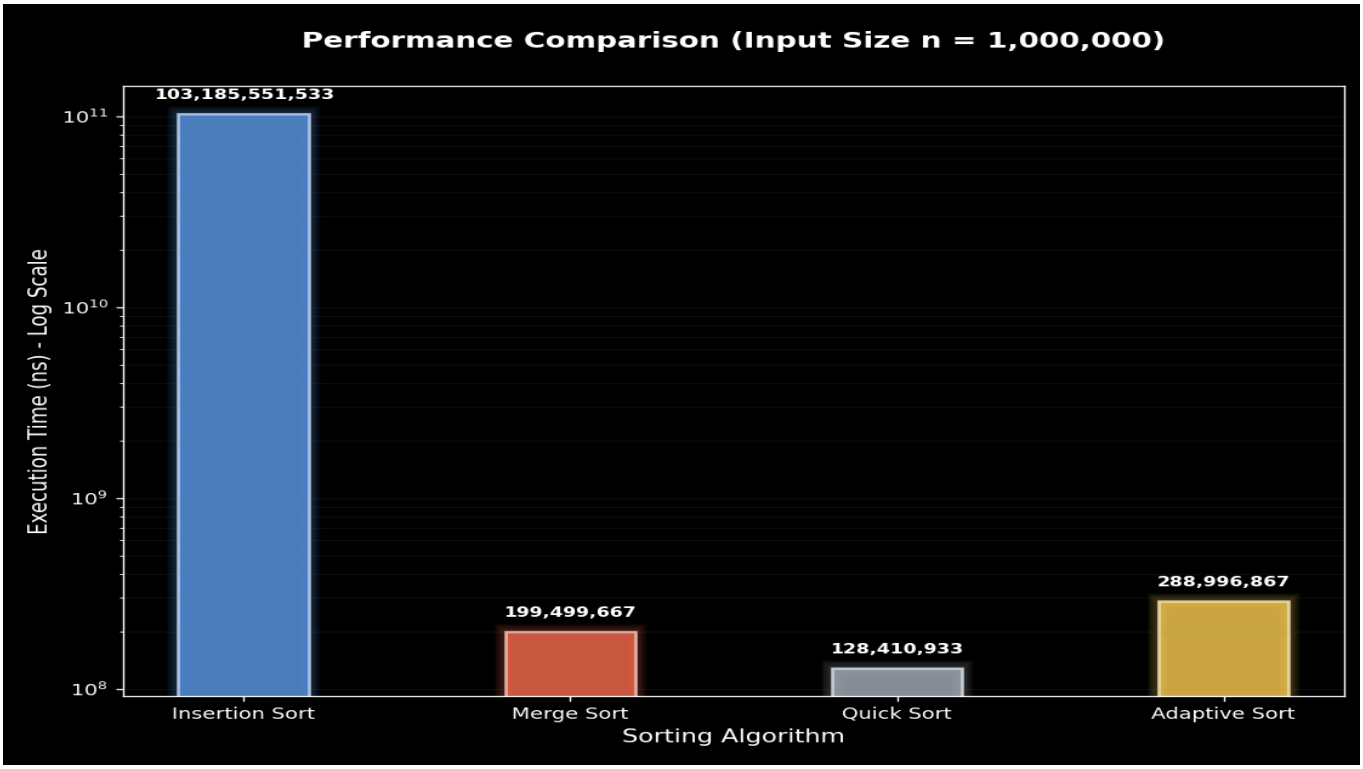


Figure 2: Execution time comparison for input size n = 1,000,000 (log scale).



DISCUSSION:

The experimental results clearly demonstrate the performance differences among the evaluated sorting algorithms as the input size increases. The observed trends closely align with the theoretical time complexity of each algorithm**.**

Insertion Sort shows a rapid increase in execution time for larger input sizes, confirming its quadratic time complexity $O(n^2)$. While it performs reasonably well for small datasets, its performance degrades significantly for large inputs, making it impractical for real-world applications involving large data volumes.

Merge Sort and Quick Sort exhibit much better scalability. Both algorithms demonstrate approximately $O(n\log\ n)$behavior, which is evident from their relatively moderate growth in execution time across increasing input sizes. Among the two, Quick Sort consistently outperforms Merge Sort for randomly distributed data, primarily due to lower constant factors and better cache utilization.

The proposed ***Adaptive Sorting algorithm dynamically selects a suitable sorting strategy based on input characteristics. Although this approach introduces a small overhead due to input analysis and decision-making, it successfully avoids the worst-case behaviour of individual algorithms***. As a result, its performance remains competitive across all tested input sizes, especially for large datasets.

For very large inputs (n = 1,000,000), the adaptive approach performs slightly slower than Quick Sort but significantly better than Insertion Sort. This trade-off is acceptable in scenarios where robustness and consistent performance are preferred over absolute speed in specific cases.

Overall, the ***results indicate that while Quick Sort is the fastest for random data, the Adaptive Sorting algorithm provides a balanced and reliable alternative, particularly in environments where input characteristics may vary and worst-case performance*** must be avoided.

CONCLUSION:

This work presented an experimental analysis of an Adaptive Sorting algorithm and compared its performance with traditional sorting techniques such as Insertion Sort, Merge Sort, and Quick Sort. The evaluation was conducted using varying input sizes, and execution time was measured to study scalability and efficiency.

The ***results show that Insertion Sort performs well only for small datasets but becomes inefficient for large inputs due to its quadratic time complexity. Merge Sort and Quick Sort demonstrated better scalability, with Quick Sort achieving the lowest execution time for random data distributions***.

The ***Adaptive Sorting algorithm effectively selected appropriate sorting strategies based on input characteristics, thereby avoiding poor performance in unfavourable cases***. Although the adaptive approach introduces additional overhead due to input analysis, it provides consistent and reliable performance across different dataset sizes. This makes it suitable for applications where input patterns are unpredictable and worst-case behaviour must be minimized.

Overall, the study confirms that adaptive sorting techniques can offer a practical balance between efficiency and robustness compared to relying on a single sorting algorithm.

FUTURE WORK:

The current implementation of the Adaptive Sorting algorithm can be further enhanced in several ways. Future work may include incorporating additional input characteristics such as partial sortedness, data distribution, and memory constraints to improve algorithm selection accuracy.

The adaptive framework can also be extended to include other advanced sorting algorithms such as Heap Sort, Tim Sort, or hybrid approaches. Additionally, testing the algorithm on real-world datasets and multi-threaded environments could provide deeper insights into its practical performance.

Further optimization of the decision-making process may help reduce overhead and improve execution time, making the adaptive approach more efficient for large-scale applications.

REFERENCE:

[1]. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, ***Introduction to Algorithms, 3rd ed., MIT Press, 2009***.
[2]. R. Sedgewick and K. Wayne, ***Algorithms, 4th ed., Addison-Wesley, 2011.***
[3]. D. E. Knuth, ***The Art of Computer Programming, Volume 3: Sorting and Searching,*** Addison-Wesley, 1998.
[4]. J. Bentley and M. McIlroy, ***"Engineering a Sort Function,"*** *Software: Practice and Experience*, vol. 23, no. 11, pp. 1249–1265, 1993.
[5]. Oracle Corporation, ***"Java Platform, Standard Edition Documentation,"*** https://docs.oracle.com/javase/

APPENDIX:

The source code for the Adaptive Sorting algorithm and benchmarking framework is available at: https://github.com/Abhi0505-kinagi/adaptive-sorting-analysis.