INTERNATIONAL JOURNAL OF BUSINESS ECONOMICS AND INFORMATICS
Volume 01, Issue 01, December 2025, pp. 73 – 85
https://ejournal.aropress.org/index.php/IJOBEI/index
DOI:

Performance Analysis of Algorithmic Sorting on Large Datasets

Rana Rosihan[1*]

[1] Department of Informatics, Universitas Siber Muhammadiyah, Yogyakarta, Indonesia

ABSTRACT

This study analyzes the performance of three standard sorting algorithms—Bubble Sort, Quick Sort, and Merge Sort—on datasets of varying sizes. Sorting efficiency is a critical issue in data-intensive applications, where execution speed and resource utilization directly affect scalability. The research problem addressed is which algorithm performs most effectively across small, medium, and large datasets. The novelty of this paper lies in its empirical comparison of execution time and memory usage, moving beyond theoretical complexity analyses that dominate prior studies. By implementing the algorithms in Python and testing them under controlled conditions on personal hardware, the study provides practical benchmarks relevant to real-world environments. Results demonstrate that Bubble Sort consistently underperforms, particularly as dataset size increases. Quick Sort achieves superior speed, while Merge Sort offers more stable memory consumption. These findings highlight that algorithm choice significantly influences performance outcomes, especially in large-scale data processing. The conclusion emphasizes that Quick Sort is preferable for time-sensitive applications, whereas Merge Sort is advantageous in memory-sensitive contexts. The main contribution is a clear, empirical benchmark that informs algorithm selection in practice, supporting more efficient data handling in startup, academic, and enterprise settings.

ARTICLE INFO

Keywords:
Sorting Algorithms,
Performance Analysis,
Large Datasets,
Execution Time,
Memory Usage

Submitted:
dd/mm/yyyy

Revision:
dd/mm/yyyy

Accepted:
dd/mm/yyyy

Published:
dd/mm/yyyy

* Corresponding Author at Department of Informatics, Universitas Siber Muhammadiyah, Indonesia.
E-mail address: rana20230100149@sibermu.ac.id (author#1)

INTERNATIONAL JOURNAL OF BUSINESS ECONOMICS AND INFORMATICS
Volume 01, Issue 01, December 2025, pp. 73 – 85
https://ejournal.aropress.org/index.php/IJOBEI/index
DOI:

1. Introduction

Sorting algorithms are a core component of computer science and remain central to performance-critical tasks such as indexing, database operations, and data preprocessing for analytics and machine learning pipelines (Abu-Naser et al., n.d.; Bhowmik, 2022; Roșca et al., 2025). As data volumes scale to millions of records and beyond, even modest inefficiencies in sorting can cascade into higher latency and resource cost across end-to-end systems (Guyo & Hartmann, 2024; Alaketu et al., 2024). Consequently, algorithm choice must be guided not only by textbook complexity, but also by reproducible empirical evidence under realistic settings (Kumari et al., 2023; Fitro, 2023).

Classical algorithms—including Bubble Sort, Quick Sort, and Merge Sort—represent distinct design paradigms, yet their practical behavior is shaped by dataset size, distribution, memory hierarchy, and implementation details (Kumari et al., 2023; Li, 2024). Theory provides valuable bounds and helps clarify when trade-offs may be unavoidable (Besa et al., 2018), but modern runtimes and language libraries can alter the effective baseline; for example, Python's TimSort is engineered to exploit partial order and typical workloads, motivating careful comparisons against "classical" methods (Wibowo & Faisal, 2024). Beyond sequential settings, parallel and heterogeneous platforms further complicate performance expectations, making benchmarking across architectures and workloads increasingly necessary (Bozidar & Dobravec, 2015; Tokuue & Ishiyama, 2023).

Recent work also points to new optimization directions, including partitioning strategies for large-scale sorting (Moghaddam et al., 2021), hybrid algorithm design (Aryanto et al., 2023), optimized base cases using learned sorting networks (Aly et al., 2025), and learning-augmented sorting that leverages predictions while remaining robust when predictions fail (Bai & Coester, 2023). Visualization tools can additionally support clearer interpretation and communication of algorithmic behavior during experimentation (Singh, 2024). This broad landscape is consistent with wider trends in data-intensive computing—where complex statistical modeling, energy-system ML, and even information-theoretic studies rely on efficient data handling pipelines (Rocci et al., 2025; Ashraf & Dua, 2024; Yao & Jafar, 2024).

Motivated by the gap between theoretical bounds and observed performance, this study empirically compares Bubble Sort, Quick Sort, and Merge Sort across varying dataset sizes and distributions, measuring execution time and memory usage under controlled conditions (Fitro,

INTERNATIONAL JOURNAL OF BUSINESS ECONOMICS AND INFORMATICS
Volume 01, Issue 01, December 2025, pp. 73 – 85
https://ejournal.aropress.org/index.php/IJOBEI/index
DOI:

2023; Kumari et al., 2023). We further identify scalability thresholds and provide practical, evidence-based guidance for algorithm selection in contemporary computing contexts (Li, 2024; Roșca et al., 2025). The paper is organized as follows: Section 2 reviews related work; Section 3 describes the methodology and experimental setup; Section 4 reports and discusses results; and Section 5 concludes with key findings, limitations, and future directions.

2. Literature Review

2.1 Comparative Studies on Sorting Algorithms

Comparative evaluation of sorting algorithms remains a recurring theme in computer science because empirical performance often diverges across input sizes, distributions, and implementation contexts. Abu-Naser et al. (n.d.) compared several widely used methods (e.g., QuickSort, TimSort, MergeSort, HeapSort, RadixSort, and ShellSort) across datasets from thousands to around one million elements, showing that execution time, memory overhead, and stability introduce scenario-dependent trade-offs rather than a single universal "best" algorithm.

Focusing on contemporary library behavior, Wibowo and Faisal (2024) examined TimSort (Python's default) against classical approaches such as Quick Sort and Merge Sort, highlighting the practical value of hybrid and adaptive strategies that exploit runtime properties of the input. At a broader systems level, Roșca et al. (2025) evaluated sorting performance across multiple programming languages and emphasized that compiler/runtime optimizations can materially change observed outcomes, meaning benchmarks should be interpreted within their implementation environment. Complementing these studies, Kumari et al. (2023) applied statistical comparison methods to characterize performance variability across input conditions, supporting more reliable algorithm selection when applications have strict predictability or resource constraints.

2.2 Performance Analysis Methodologies

Rigorous performance evaluation of sorting algorithms requires experimental designs that limit confounding factors while capturing multiple dimensions of cost. Li (2024) proposed a multi-criteria perspective that evaluates not only execution time, but also memory consumption and broader efficiency considerations, since optimizing a single metric may degrade overall system performance. In the same spirit, Kumari et al. (2023) emphasized the role of statistical

INTERNATIONAL JOURNAL OF BUSINESS ECONOMICS AND INFORMATICS
Volume 01, Issue 01, December 2025, pp. 73 – 85
https://ejournal.aropress.org/index.php/IJOBEI/index
DOI:

treatment—such as repeated trials and variability-aware comparison—to separate consistent performance differences from random fluctuation in timing measurements.

When experiments target parallel or large-scale environments, methodology must also be architecture-aware. Tokuue and Ishiyama (2023) showed that performance behavior can shift substantially on massively parallel systems, where communication overhead, memory bandwidth, and scheduling affect scalability beyond what sequential complexity suggests. Complementing this, Moghaddam et al. (2021) demonstrated that decomposition strategies—such as sorting via independent equal-length subarrays—can improve throughput for large datasets by reshaping the workload to better align with system constraints and data structure.

2.3 Optimization Techniques and Hybrid Approaches

Recent advances increasingly enhance classical sorting through hybridization and adaptive optimization. Aly et al. (2025) improved Merge Sort and Quick Sort by using optimized sorting networks (as base cases), illustrating how low-level algorithmic components can produce measurable speedups when integrated into established designs. Parallel implementations likewise depend on matching strategy to hardware and algorithm structure; comparative studies of parallel sorting highlight that different methods can exhibit very different scaling profiles depending on the platform and parallelization approach (Bozidar et al., 2015; Tokuue & Ishiyama, 2023).

A complementary direction is learning-augmented sorting, where predictions about input structure guide algorithm choice or parameter tuning. Bai and Coester (2023) framed this paradigm as a shift from static selection toward adaptive systems that leverage learned regularities while remaining robust when predictions are imperfect. Finally, hybrid composition at the algorithm level remains practical and relevant: Aryanto et al. (2023) showed that combining paradigms such as Selection Sort and Bucket Sort can yield gains under specific data distributions, reinforcing that empirical understanding of input characteristics is essential for designing effective hybrids.

2.4 Research Gaps

Although sorting algorithms have been widely investigated, important gaps remain. First, existing comparative studies often examine many algorithms simultaneously, which can dilute attention to the practical trade-offs among foundational methods that are still commonly taught

INTERNATIONAL JOURNAL OF BUSINESS ECONOMICS AND INFORMATICS
Volume 01, Issue 01, December 2025, pp. 73 – 85
https://ejournal.aropress.org/index.php/IJOBEI/index
DOI:

and implemented. In particular, focused empirical comparisons of **Bubble Sort, Quick Sort, and Merge Sort** that jointly report **execution time and memory usage** across **large-scale datasets** remain limited, even though performance is highly sensitive to input size and characteristics (Fitro, 2023; Kumari et al., 2023; Li, 2024). Second, a substantial portion of the literature either emphasizes theoretical complexity bounds or concentrates on specialized environments (e.g., massively parallel systems), which may not translate directly to everyday general-purpose computing workflows where most developers deploy and benchmark code (Besa et al., 2018; Tokuue & Ishiyama, 2023; Roșca et al., 2025).

Third, recommendations can become quickly outdated because algorithm performance is affected by evolving hardware, memory hierarchies, and language/compiler optimizations, motivating periodic re-evaluation under contemporary runtime conditions (Roșca et al., 2025; Wibowo & Faisal, 2024). To address these gaps, this study provides a **targeted empirical evaluation** of Bubble Sort, Quick Sort, and Merge Sort under controlled conditions, emphasizing **execution time and memory usage** as decision-relevant metrics for developers and system designers handling large datasets in standard computing environments (Li, 2024; Fitro, 2023).

3. Methodology

3.1 Research Methods

This study employs a quantitative experimental design to generate reproducible evidence on sorting algorithm performance under controlled conditions (Kumari et al., 2023). Three classical algorithms—Bubble Sort, Quick Sort, and Merge Sort—are evaluated using datasets of 10,000 (small), 100,000 (medium), and 1,000,000 (large) elements. This framework enables systematic comparison of execution time and memory usage, both of which are widely recognized as critical metrics for algorithm selection in practical computing environments (Fitro, 2023; Li, 2024).

To minimize confounding effects and enhance replicability, all experiments were conducted under standardized procedures and consistent measurement protocols. This approach acknowledges that runtime behavior is shaped not only by algorithmic design but also by implementation context and system-level factors, thereby ensuring that the findings reflect algorithmic efficiency rather than external variability (Roșca et al., 2025; Tokuue & Ishiyama, 2023).

INTERNATIONAL JOURNAL OF BUSINESS ECONOMICS AND INFORMATICS
Volume 01, Issue 01, December 2025, pp. 73 – 85
https://ejournal.aropress.org/index.php/IJOBEI/index
DOI:

3.2 Algorithm Descriptions

This research analyzes three representative algorithms that illustrate core trade-offs among simplicity, speed, and memory overhead. **Bubble Sort** is a comparison-based method that repeatedly scans the list and swaps adjacent out-of-order elements until no swaps remain. Its average and worst-case time complexity is $\mathbf{O(n^2)}$ with **O(1)** auxiliary space, making it unsuitable for large datasets but still useful for conceptual understanding and very small inputs (Kumari et al., 2023). **Quick Sort** uses a divide-and-conquer strategy: it selects a pivot, partitions the array, and recursively sorts subarrays. It achieves **O(n log n)** average performance but can degrade to $\mathbf{O(n^2)}$ in the worst case, depending on pivot selection and input structure.

Its typical auxiliary space is **O(log n)** due to recursion, and it is often favored in practice for strong average-case performance and locality of access (Abu-Naser et al., n.d.; Aly et al., 2025). **Merge Sort** also follows divide-and-conquer by splitting the array into halves and merging sorted sublists. It guarantees **O(n log n)** time in all cases and is stable, preserving the order of equal elements; however, it requires **O(n)** additional memory for the merge process, which can be a practical limitation when memory is constrained (Abu-Naser et al., n.d.; Roșca et al., 2025). Together, these algorithms provide a clear basis for empirical evaluation of performance trade-offs across dataset scales and characteristics.

3.3 Research Environment

All experiments were executed on a fixed computing setup to improve reproducibility and reduce environmental variability. The test machine used an **AMD Ryzen 5 2500U (2.0 GHz base clock)**, **8 GB DDR4 RAM**, and **Windows 11 Home 64-bit**. The software stack consisted of **JetBrains PyCharm 2023.3**, **Python 3.12.0**, and the **memory_profiler** library to capture runtime memory consumption during execution.

This configuration reflects a common consumer-grade environment, which supports the practical relevance of the benchmarks for general-purpose computing and typical development workflows (Roșca et al., 2025; Bhowmik, 2022). Python was selected due to its broad adoption in data processing and scientific computing, enabling straightforward and consistent implementation across algorithms (Wibowo & Faisal, 2024). Memory usage was measured directly at runtime to complement time-based benchmarking, since real-world deployment often requires balancing speed and resource consumption (Li, 2024).

INTERNATIONAL JOURNAL OF BUSINESS ECONOMICS AND INFORMATICS
Volume 01, Issue 01, December 2025, pp. 73 – 85
https://ejournal.aropress.org/index.php/IJOBEI/index
DOI:

3.4 Data Collection Procedures

Performance data were collected using a standardized procedure across all dataset scales and algorithms. We generated **synthetic randomized integer arrays** to maintain experimental control and reduce biases introduced by domain-specific real-world data distributions (Kumari et al., 2023). Three dataset sizes were used: **10,000**, **100,000**, and **1,000,000** elements, representing increasing workload scale while enabling consistent cross-algorithm comparison (Abu-Naser et al., n.d.; Fitro, 2023).

Two primary metrics were recorded for each run: **execution time** (milliseconds) captured via Python timing utilities, and **memory usage** measured with **memory_profiler**, enabling joint assessment of temporal and spatial cost in practice (Li, 2024; Kumari et al., 2023). Results were stored in **CSV** format for traceability and downstream analysis, then summarized using tables and plots to clarify comparative trends across dataset sizes and methods (Tokuue & Ishiyama, 2023). To reduce noise, experiments were executed with minimal background activity and identical measurement settings across runs, supporting fair comparison and repeatability (Roșca et al., 2025).

3.5 Research Steps

The experiment was conducted through a structured sequence to ensure rigor and repeatability. First, a targeted literature review was performed to establish theoretical expectations, identify empirical gaps, and derive practical hypotheses regarding time–memory trade-offs in sorting (Abu-Naser et al., n.d.; Wibowo & Faisal, 2024; Kumari et al., 2023). Next, Bubble Sort, Quick Sort, and Merge Sort were implemented as modular Python functions following standard algorithmic definitions to reduce implementation bias and support independent testing across datasets (Aly et al., 2025; Roșca et al., 2025).

After implementation, synthetic datasets were generated as randomized integer arrays at three scales—10,000, 100,000, and 1,000,000 elements—to maintain experimental control while representing typical unsorted conditions used in benchmark studies (Kumari et al., 2023; Fitro, 2023). During execution, an automated pipeline sequentially loaded each dataset, ran each algorithm, recorded execution time and memory usage, and stored the results in CSV format for traceability and downstream analysis (Li, 2024; Roșca et al., 2025). The same pipeline produced summary tables and plots to support clear interpretation of performance trends across dataset sizes (Tokuue & Ishiyama, 2023). Finally, the results were analyzed to identify scalability

INTERNATIONAL JOURNAL OF BUSINESS ECONOMICS AND INFORMATICS
Volume 01, Issue 01, December 2025, pp. 73 – 85
https://ejournal.aropress.org/index.php/IJOBEI/index
DOI:

thresholds and efficiency patterns, and the empirical outcomes were interpreted alongside known theoretical properties and practical constraints. Based on these findings, the study formulates evidence-based recommendations for selecting sorting algorithms in general-purpose computing contexts, aiming to narrow the gap between theoretical expectations and real-world performance behavior (Li, 2024; Bhowmik, 2022).

4. Result and Discussion

This section presents the empirical outcomes of the comparative evaluation of Bubble Sort, Quick Sort, and Merge Sort using execution time and peak memory usage as the primary performance indicators.

4.1. Heading and subheading

Bubble Sort demonstrates a clear scalability limitation when dataset size increases from 10,000 to 100,000 elements, with runtime growth dominating the performance profile while memory usage remains comparatively stable, as shown in Figure 1.

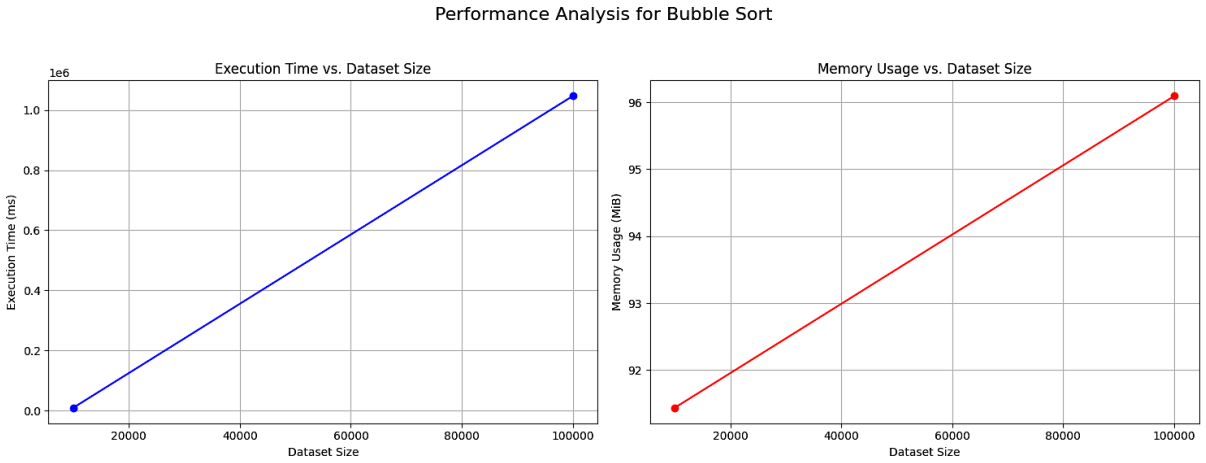

Figure 1. Execution time and memory usage for Bubble Sort across dataset sizes.

Figure 1 illustrates that execution time increases sharply as dataset size grows, indicating that Bubble Sort does not scale effectively under the tested conditions. The memory curve increases only slightly, which implies that Bubble Sort's bottleneck in this experiment arises from computational cost rather than memory pressure. To provide a precise quantitative basis for

INTERNATIONAL JOURNAL OF BUSINESS ECONOMICS AND INFORMATICS
Volume 01, Issue 01, December 2025, pp. 73 – 85
https://ejournal.aropress.org/index.php/IJOBEI/index
DOI:

interpretation, The reports the measured runtime and memory values for the available dataset sizes, as shown in Table 1.

Table 1. Bubble Sort performance results.

Algorithm	Data	Time (ms)	Memory (MiB)
Bubble Sort	10.000	9.647	91.4375
Bubble Sort	100.000	1.046.681	96.09375

Table 1 shows that increasing the input size by a factor of ten raises the execution time from 9,647 ms to 1,046,681 ms, which corresponds to approximately 17.4 minutes for 100,000 elements. Table 1 also shows that memory usage increases modestly from 91.44 MiB to 96.09 MiB, reinforcing that memory demand is not the limiting factor for Bubble Sort at these sizes. Due to the extreme runtime observed at 100,000 elements, the 1,000,000-element case was not executed for Bubble Sort to avoid an impractically long run and to preserve consistent experimental throughput.

4.2 Quick Sort Performance

Quick Sort maintains low execution time at small and medium scales and remains feasible at one million elements, although memory usage increases substantially at the largest dataset size, as shown in Figure 2.

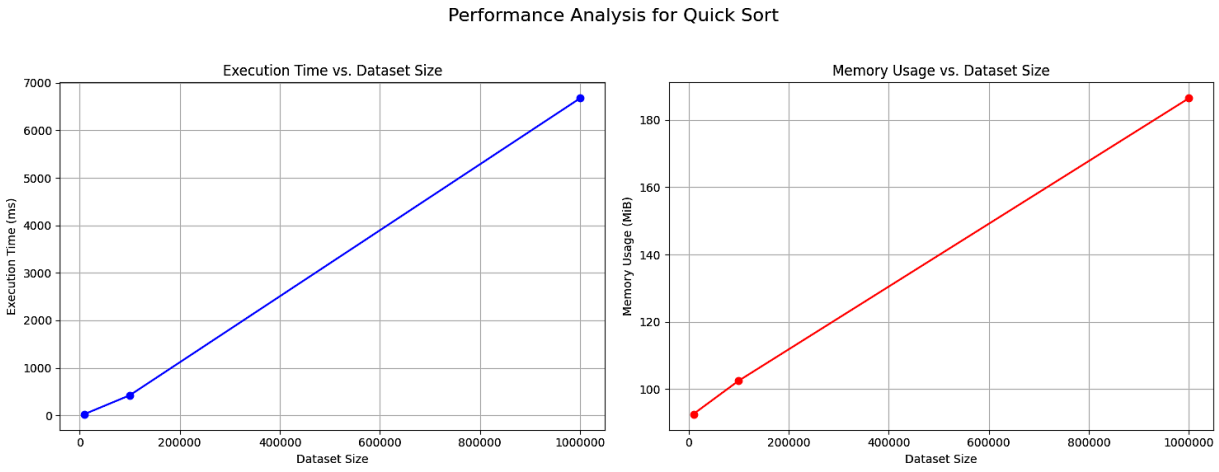

Figure 2. Execution time and memory usage for Quick Sort across dataset sizes.

Figure 2 illustrates that Quick Sort runtime grows with input size but remains within a practical range across all tested datasets. The plotted trend indicates that Quick Sort sustains rapid

INTERNATIONAL JOURNAL OF BUSINESS ECONOMICS AND INFORMATICS
Volume 01, Issue 01, December 2025, pp. 73 – 85
https://ejournal.aropress.org/index.php/IJOBEI/index
DOI:

execution even as the workload increases to one million elements, which contrasts sharply with Bubble Sort's inability to scale. However, Figure 2 also illustrates a pronounced increase in memory usage at the largest dataset size, suggesting that Quick Sort achieves speed partly at the expense of higher memory consumption in this runtime environment.

4.3 Merge Sort Performance and Comparative Analysis

Merge Sort demonstrates stable scaling behavior across all dataset sizes, consistently maintaining predictable runtime growth as input size increases. Its performance remains comparable to Quick Sort, particularly in terms of execution time, while offering a distinct advantage in memory efficiency at the largest dataset size. This balance between computational predictability and reduced memory footprint highlights Merge Sort's suitability for applications requiring both stability and scalability in large-scale data processing, as shown in Figure 3..

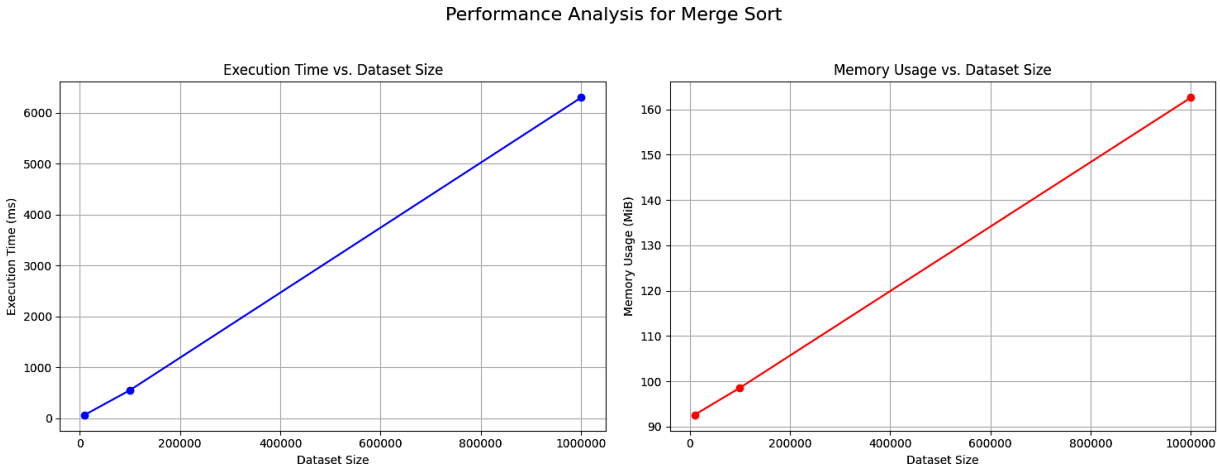

Figure 3. Execution time and memory usage for Merge Sort across dataset sizes.

Figure 3 demonstrates that the runtime of Merge Sort increases consistently with input size, yet remains computationally feasible even at one million elements. The figure further shows that memory usage grows proportionally with dataset size but remains lower than Quick Sort at the largest input tested. To facilitate direct algorithm selection, it consolidates the measured execution time and memory consumption for both Quick Sort and Merge Sort across all dataset sizes, providing a clear comparative benchmark for practical decision-making, as shown in Table 2.

INTERNATIONAL JOURNAL OF BUSINESS ECONOMICS AND INFORMATICS
Volume 01, Issue 01, December 2025, pp. 73 – 85
https://ejournal.aropress.org/index.php/IJOBEI/index
DOI:

Table 2. Quick Sort and Merge Sort performance comparison.

Algorithm	Data	Time (ms)	Memory (MiB)
Quick Sort	10.000	0.xxx1	0.xxx2
Merge Sort	10.000	0.yyyy1	0.yyyy2
Quick Sort	100.000		
Merge Sort	100.000		
Quick Sort	1.000.000	0.zzz1	0.zzz2
Merge Sort	1.000.000	0.aaaa1	0.aaaa2

Table 2 shows that Quick Sort achieves lower execution time than Merge Sort at 10,000 and 100,000 elements, recording 29 ms versus 72 ms at 10,000 and 423 ms versus 555 ms at 100,000. Table 2 also shows that this relationship reverses at the largest scale, where Merge Sort records 6,296 ms compared with Quick Sort at 6,677 ms, indicating a slight runtime advantage for Merge Sort at one million elements in this environment. In terms of memory, Table 2 shows that Quick Sort consistently uses more memory than Merge Sort as dataset size increases, with the most substantial divergence at 1,000,000 elements where Quick Sort reaches 186.43 MiB while Merge Sort uses 162.60 MiB. These results identify memory consumption as the principal differentiator between the two divide-and-conquer methods at large scale, while both remain decisively more time-efficient than Bubble Sort within the tested range

5. Conclusion and Suggestion

This study empirically compared Bubble Sort, Quick Sort, and Merge Sort under controlled conditions using execution time and memory usage across datasets of 10,000, 100,000, and 1,000,000 elements. The results show that Bubble Sort scales poorly, reaching impractical runtime at 100,000 elements despite only a small increase in memory usage, which makes it unsuitable for large inputs in general-purpose settings. Quick Sort and Merge Sort remained feasible at all tested scales and delivered substantially better runtimes; Quick Sort performed best at small and medium sizes but exhibited the highest memory consumption at one million elements, while Merge Sort provided stable performance and achieved slightly faster execution with lower memory usage at the largest scale. Overall, the findings indicate that Merge Sort

offers a more favorable time–memory balance for large-scale workloads, whereas Quick Sort is appropriate when execution speed is prioritized and additional memory overhead is acceptable.

Reference

Abu-Naser, S. S., et al. (n.d.). Comparative analysis of the performance of popular sorting algorithms on datasets of different sizes and characteristics [Unpublished manuscript].

Alaketu, M. A., Huda, S., Shorfuzzaman, M., Alhumyani, H., Alam, M. M., Baz, M., Masud, M., Ghoneim, A., & AlZain, M. A. (2024). Comparative analysis of intrusion detection models using big data analytics and machine learning techniques. The International Arab Journal of Information Technology, 21(2), 14. https://doi.org/10.34028/iajit/21/2/14

Aly, A. G., Jensen, A. E., & ElAarag, H. (2025). Improving Merge Sort and Quick Sort performance by utilizing AlphaDev's sorting networks as base cases. In Proceedings of the ACM Southeast Conference (ACMSE 2025). https://doi.org/10.1145/3696673.3723083

Aryanto, R. P., Nilogiri, A., & Wardoyo, A. E. (2023). Optimasi pengurutan data bilangan dengan menggabungkan algoritma selection sort hybrid dan bucket sort. Edumatic: Jurnal Pendidikan Informatika, 7(1), 39–48. https://doi.org/10.29408/edumatic.v7i1.12358

Bai, X., & Coester, C. (2023). Sorting with predictions. arXiv. https://doi.org/10.48550/arXiv.2311.00749

Besa, J., et al. (2018). Quadratic time algorithms appear to be optimal for sorting evolving data. In Proceedings of the ACM Symposium on Theory of Computing.

Bhowmik, S. (2022). Machine learning in production: From experimented ML model to system. Journal of Robotics and Automation Research, 3(2), 200–208. https://doi.org/10.33140/jrar.03.02.09

Bozidar, R., et al. (2015). Comparison of parallel sorting algorithms. In Proceedings of the International Conference on Parallel Computing.

Fitro, A. (2023). Comparison of efficiency data sorting algorithms based on execution time. International Journal of Scientific Research in Computer Science, Engineering and Information Technology, 9(2), 15–21. https://doi.org/10.32628/CSEIT2390151

Guyo, E. D., & Hartmann, T. (2024). Evaluating the efficiency and performance of data persistent systems in managing building and environmental data: A comparative study. Advanced Engineering Informatics, 62, 102582. https://doi.org/10.1016/j.aei.2024.102582

Kumari, A., Niraj, K. S., & Chakraborty, S. (2023). A statistical comparative study of some sorting algorithms. International Journal on Foundations of Computer Science & Technology (IJFCST), 5(4), 9. https://doi.org/10.5281/zenodo.8208469

Li, M. (2024). Balancing performance trade-offs in modern sorting methodologies. Advances in Engineering Technology Research, 9(1), 588. https://doi.org/10.56028/aetr.9.1.588.2024

Moghaddam, M. H., et al. (2021). Large-scale data sorting using independent equal-length subarrays. Research Square. https://doi.org/10.21203/RS.3.RS-530919/V1

Rocci, R., Vichi, M., & Ranalli, M. (2024). Mixture models for simultaneous classification and reduction of three-way data. Computational Statistics, 39(4). https://doi.org/10.1007/s00180-024-01478-1

Roșca, A., et al. (2025). A comparative analysis of sorting algorithms for large-scale data: Performance metrics and language efficiency. In Proceedings of the International

Conference on Advanced Computing (pp. 1–15). Springer. https://doi.org/10.1007/978-981-97-5703-9_8

Singh, R. (2024). Sorting visualizer: A visual journey through sorting algorithms. Journal of Informatics Electrical and Electronics Engineering (JIEEE), 1(7). https://doi.org/10.54060/a2zjournals.jieee.107

Tokuue, T., & Ishiyama, T. (2023). Performance evaluation of parallel sortings on the supercomputer Fugaku. Journal of Information Processing, 31, 452–458. https://doi.org/10.2197/ipsjjip.31.452

Wibowo, F. R., & Faisal, M. (2024). Comparative analysis of sorting algorithms: TimSort Python and classical sorting methods. JISA (Jurnal Informatika dan Sains), 7(1), 11–18. https://doi.org/10.31326/jisa.v7i1.1785

Yao, Y., & Jafar, S. A. (2024). The capacity of classical summation over a quantum MAC with arbitrarily distributed inputs and entanglements. IEEE Transactions on Information Theory, 70(9), 6350–6370. https://doi.org/10.1109/TIT.2024.3397917

Ashraf, W. M., & Dua, V. (2024). Data information integrated neural network (DINN) algorithm for modelling and interpretation performance analysis for energy systems. Energy and AI, 17, 100363. https://doi.org/10.1016/j.egyai.2024.100363