Comparative Analysis of Sorting Algorithm Efficiency in C++, Java, and Python

Emma R. Hoffmann

Department of Computer Science

St. Cloud State University

erhoffmann1@stcloudstate.edu

Abstract

This paper presents a comparative analysis of the efficiency of five sorting algorithms: Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, and Quick Sort, implemented across three different programming languages: C++, Java, and Python. Concentrating on datasets that vary in size from 1,000, 10,000, and 100,000 elements, the study examines the efficiency of these algorithms across different dataset sizes to determine the influence of programming language selection on algorithm performance. Performance metrics were collected through rigorous experimentation, utilizing language-specific timing functions to ensure accuracy. Findings suggest significant variations in execution time across languages and algorithms, highlighting the influence of language constructs and runtime environments on sorting efficiency. This study contributes to the understanding of algorithmic performance in varied programming contexts, providing critical guidance for the strategic selection of programming languages and algorithms in data-intensive applications.

Introduction

Efficient data sorting stands at the core of computer science. It is pivotal for the functionality and performance of a broad range of applications. This study presents a comparative analysis of five widely recognized sorting algorithms across three of the most prevalent programming languages. It aims to evaluate how the choice of programming language influences algorithm performance, particularly focusing on execution times across varying scales of data.

The examination of sorting efficiency transcends theoretical discussion, addressing practical implications for technology's speed and reliability. Drawing upon seminal literature and employing rigorous experimental procedures, this paper evaluates each algorithm's performance, emphasizing the nuanced relationship between algorithmic design and programming language structure.

This research aims to equip students, researchers, and developers with the insights needed for strategic language and algorithm selection and application. Through a detailed presentation of methodologies and outcomes, the paper intends to support advancements in software development practices, emphasizing optimization and the efficient handling of data.

Literature Review

The efficiency of sorting algorithms is fundamental in computational theory and practice, with considerable implications for database management and artificial intelligence. Performance optimization is essential, particularly as datasets grow exponentially in size. Classical sorting algorithms such as Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, and Quick Sort are regularly benchmarked for efficiency due to their varied computational complexities and differing behaviors under diverse conditions (Al-Kharabsheh et al., 2013).

Algorithmic performance is often assessed in terms of time complexity, with a specific focus on dataset size impact. Selection Sort, despite its simplicity and $O(n^2)$ time complexity, is inefficient for large datasets due to excessive comparisons (Hayfron-Acquah & D, 2015). In contrast, Merge Sort and Quick Sort, both operating at an average time complexity of O(n log n), demonstrate superior performance for larger arrays, leveraging divide-and-conquer strategies to minimize processing time (Al-Kharabsheh et al., 2013).

The stability of sorting algorithms – which ensures the preservation of input order among equal elements – and memory overhead, particularly for recursive algorithms like Quick Sort, are crucial considerations for algorithm selection (Al-Kharabsheh et al., 2013). Empirical studies suggest that while Quick Sort generally offers the fastest execution times for sizable arrays, algorithms like Selection Sort underperform as data scales increase, emphasizing the significance of algorithm selection relative to the problem domain (Durrani, 2015).

The iterative improvement of sorting algorithms focuses on refining existing ones while influencing the design of future, more efficient algorithms. The persistent evaluation of sorting algorithms supports the dynamic and evolving nature of computational efficiency research (Sareen, 2013).

This review sets the context for analyzing how programming languages — C++, Java, and Python — impact the performance of these foundational sorting algorithms. As Fangohr asserts, it aims to deepen the understanding of the relationship between programming language constructs, runtime environments, and their combined effect on algorithmic efficiency (Fangohr, 2004).

Implemented Sorting Algorithms

This section examines five specific sorting algorithms: Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, and Quick Sort. Each of these algorithms demonstrates a unique approach to sorting, ranging from straightforward, linear methods to more nuanced and strategic techniques. Through a variety of algorithms, a comprehensive overview of the diverse methodologies in computational sorting is obtained.

1. Selection Sort

Selection Sort is a simple sorting method that iteratively locates the smallest element in an array and relocates it to the beginning. This procedure is reiterated on the remaining unsorted section of the array.

While Selection Sort maintains a consistent time complexity of $O(n^2)$ across all scenarios (Best, Average, and Worst), it exhibits this behavior because it continues sorting even when the remaining array elements are already sorted.

Though Selection Sort is uncomplicated to implement and advantageous for sorting small datasets with minimal memory usage, it proves inefficient for larger lists due to its quadratic time complexity.

2. Bubble Sort

Bubble Sort is a sorting technique that compares adjacent elements in an array and swaps them if they are in the incorrect order. This process is repeated until the entire array is sorted.

In the best case scenario, which is when the data is already sorted, Bubble Sort has a time complexity of $O(n)$. However, in the average and worst cases, it has a time complexity of $O(n^2)$ as it requires at least two passes through the data.

Despite Bubble Sort's simplicity and ease of implementation, it is considered inefficient for large datasets.

3. Insertion Sort

Insertion Sort is a method that builds the sorted array one element at a time. It starts with a single element (the first element) and iteratively inserts each subsequent element into its correct position within the sorted portion of the array.

Similar to Bubble Sort, Insertion Sort has a time complexity of $O(n^2)$ in the average and worst cases. In the best case, Insertion Sort has a time complexity of $O(n)$ when the data is already sorted.

Insertion Sort is efficient for small datasets but becomes less efficient as the dataset size increases.

4. Merge Sort

Merge Sort is a divide-and-conquer sorting algorithm that divides the array into smaller sub-arrays, recursively sorts them, and then merges them to produce the final sorted array.

Merge Sort consistently exhibits a time complexity of $O(n \log n)$ across all scenarios.

It is known for its efficiency and stability in sorting. However, Merge Sort requires additional memory for storing the sub-arrays, which can be a drawback in memory-constrained environments.

5. Quick Sort

Quick Sort is a sorting method that selects a pivot element, partitions the array into two sub-arrays (elements less than the pivot and elements greater than the pivot), and then recursively sorts these sub-arrays.

On average, Quick Sort has a time complexity of $O(n \log n)$, making it efficient. However, in the worst case, it can downgrade to $O(n^2)$ if poorly chosen pivot elements lead to unbalanced partitions.

Quick Sort is widely used but may consume additional memory due to its recursive nature.

Methodology and Experimental Setup

The execution time performance of the five implemented sorting algorithms was conducted on a computer running Windows 11 Pro 64-bit operating system having Intel Core i7 and 16GB installed RAM. For the development environment, Visual Studio Code running version 1.86.0 was utilized.

The experiment involved the use of three programming languages: C++, Java, and Python. Timing functions specific to each programming language were employed for measuring execution time: C++ - ‘std::chrono’, Java - ‘System.nanoTime()’, Python - ‘time.time()’.

For the experimentation three CSV files were prepared, each containing random numerical elements within the range of -1000 to 1000. The datasets consisted of 1000, 10,000, and 100,000 random elements. The randomized dataset files remained consistent throughout the experiment. Essentially, the content for each dataset size was unchanged for experimentation across all algorithms and languages. To ensure reliability in the analysis, each dataset was subjected to 10 trials for each of the five sorting algorithms. The results of these 10 trials were averaged to a single average for each sorting algorithm and dataset combination.

To conclude, the experiment resulted in a total of 45 distinct performance outcomes.

Results and Analysis

This section outlines the execution times for five sorting algorithms across C++, Java, and Python, illustrating how each programming language's operational efficiency impacts algorithm performance. The data enables a comparative analysis that not only highlights the efficiency of different algorithms within each language but also offers insight into how these efficiencies scale with increased data complexity.

C++ Performance Analysis

In C++, Quick Sort consistently outperforms the other algorithms across all dataset sizes, showcasing its efficiency with an execution time of merely 0.07 ms for 1,000 elements and scaling impressively to 14.13 ms for 100,000 elements (Table 1). Merge Sort also demonstrates robust performance, particularly notable in handling larger datasets with a time of 97.31 ms for 100,000 elements, indicating its effectiveness in divide-and-conquer strategies. Conversely, Bubble Sort and Selection Sort show significantly higher execution times, particularly at larger scales, highlighting their inefficiencies in handling bulk data due to their quadratic time complexity.

Table 1: ***Execution Times in C++ for Various Dataset Sizes***

Dataset Size	Selection Sort	Bubble Sort	Insertion Sort	Merge Sort	Quick Sort

	(ms)	(ms)	(ms)	(ms)	(ms)
1,000	1.53	2.66	1.42	0.74	0.07
10,000	161.21	321.03	107.09	7.91	0.99
100,000	15905.42	35161.61	10751.40	97.31	14.13

Java Performance Analysis

In Java, Quick Sort and Merge Sort again stand out for their performance, with Quick Sort completing in 0.28 ms for 1,000 elements and Merge Sort showing exceptional scalability with 25.68 ms for 100,000 elements (Table 2). This suggests that Java's runtime environment and garbage collection mechanisms do not significantly hinder these algorithms' divide-and-conquer efficiency. However, as in C++, Bubble Sort and Selection Sort lag considerably in larger datasets, highlighting their limited applicability in performance-critical applications.

Table 2: ***Execution Times in Java for Various Dataset Sizes***

Dataset Size	Selection Sort (ms)	Bubble Sort (ms)	Insertion Sort (ms)	Merge Sort (ms)	Quick Sort (ms)
1,000	2.39	4.81	2.11	0.56	0.28
10,000	116.94	226.01	53.59	5.46	1.91
100,000	8844.35	41715.34	6774.09	25.68	18.59

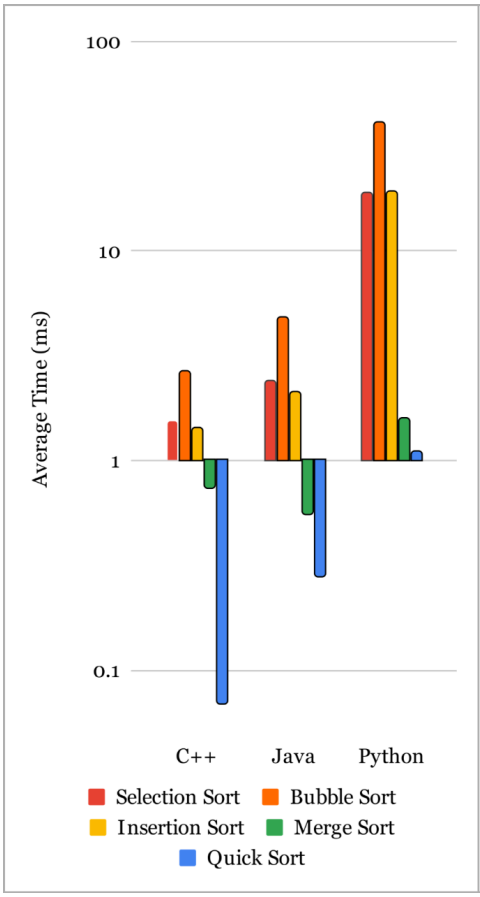
Python Performance Analysis

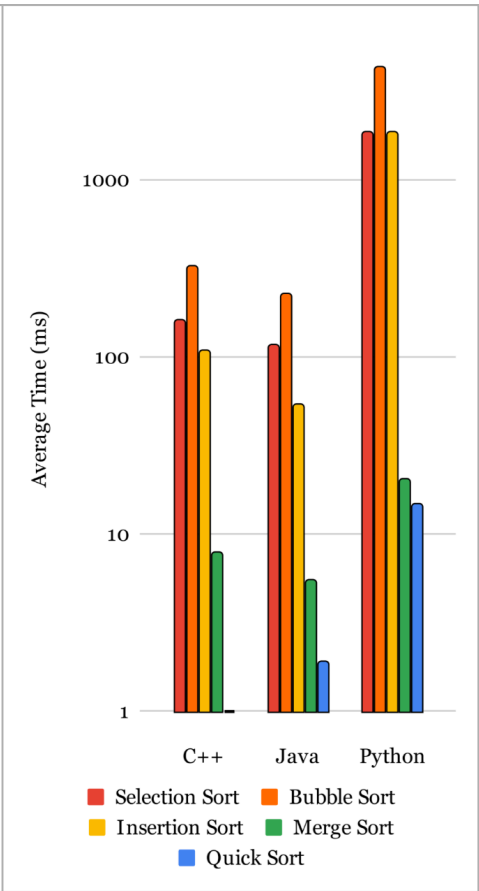
Python's results highlight the interpreted language's performance penalty compared to C++ and Java. For instance, Quick Sort takes 1.10 ms for 1,000 elements and escalates to 245.42 ms for 100,000 elements, a marked increase from the compiled languages (Table 3). This trend is consistent across all algorithms, with Selection Sort and Bubble Sort showing excessively long execution times for the largest dataset. Nonetheless, the relative performance rankings among the algorithms remain consistent, with Quick Sort and Merge Sort outperforming others, emphasizing the algorithms' inherent efficiencies despite Python's slower execution speeds.

Table 3: ***Execution Times in Python for Various Dataset Sizes***

Dataset Size	Selection Sort (ms)	Bubble Sort (ms)	Insertion Sort (ms)	Merge Sort (ms)	Quick Sort (ms)
1,000	18.96	41.24	19.05	1.60	1.10
10,000	1846.85	4310.98	1827.35	20.15	14.78
100,000	179778.96	435743.07	188058.49	224.19	245.42

Comparative Performance Analysis Across Languages and Dataset Sizes





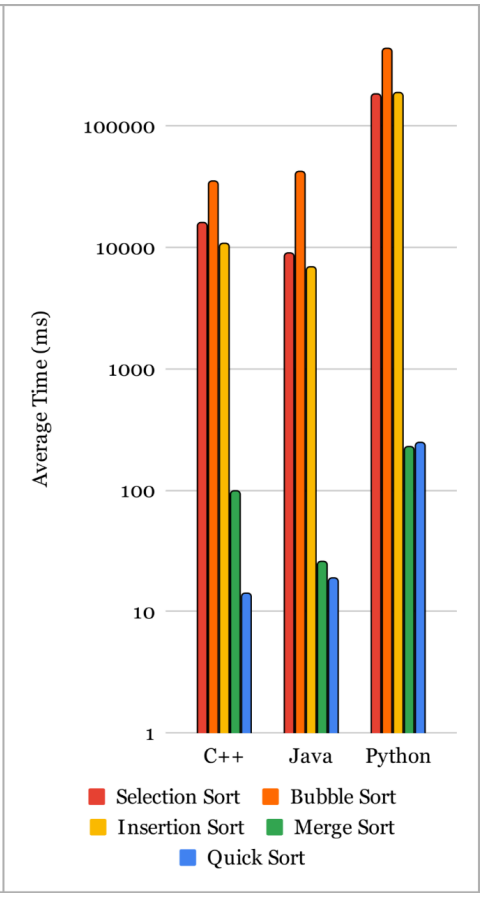


Figure 1: ***1,000 Elements*** **Figure 2:** ***10,000 Elements*** **Figure 3:** ***100,000 Elements***

Figures 1, 2, 3: ***Execution Times for Various Dataset Sizes***

The comparative performance analysis across Figures 1, 2, and 3 provides a nuanced understanding of how sorting algorithm efficiency scales with increasing dataset sizes. Across these datasets remains a consistent pattern. Each figure outlines how efficiency changes with dataset sizes that increase by an order of magnitude, with each subsequent size ten times the previous one. The graphs compare the performance of sorting algorithms for datasets of 1,000, 10,000, and 100,000 elements, highlighting the intricate dynamics of algorithm efficiency in relation to increasing data volumes.

Figure 1 with 1,000 elements indicates Python shows significantly higher execution times for all algorithms when compared to C++ and Java, with Python's Bubble Sort being the most time-consuming. Quick Sort remains the fastest algorithm across all three languages, with C++ showcasing the shortest execution time.

As Figure 2 illustrates, the increase in dataset size presents a growing computational challenge, continuing the trend observed with the 1,000-element dataset. Python remains notably slower in performance. Interestingly, Java slightly outperforms C++ in the efficiency of most algorithms,

showcasing its handling of mid-sized datasets. Despite this, C++'s Quick Sort maintains a slight advantage. Overall, this shift signals the beginning of a noticeable trend in scalability, where Merge Sort and Quick Sort start to demonstrate efficiency in handling larger datasets. This capability is not observed with the more traditional algorithms of Bubble Sort and Selection Sort.

By Figure 3, 100,000 elements, the dramatic scaling of execution times magnificence the efficiency of Merge Sort and Quick Sort in managing large datasets. This is particularly evident in Python. Despite overall longer execution times in Python, the relative increase in execution times for Merge Sort and Quick Sort is significantly less pronounced compared to the quadratic time complexity algorithms. C++ and Java continue to showcase the most optimal execution times, with C++'s Quick Sort maintaining its position as the most efficient algorithm across all tested languages and dataset sizes.

Performance Overview and Strategic Insights

This overview consolidates the findings from our examination of sorting algorithm execution times across C++, Java, and Python, as visualized through Tables 1, 2, and 3, and Figures 1, 2, and 3. It provides insights based on comparative analysis and emphasizes the scalability of algorithms and the efficiency of programming languages with increased dataset sizes.

C++ demonstrates exceptional processing speed, particularly with Quick Sort, which consistently outperforms throughout all evaluated dataset sizes. This affirms the potent efficiency of compiled languages in managing data-intensive operations, with Quick Sort exemplifying optimal performance scalability within C++'s environment. Java reveals its strengths in specific sorting algorithms. Java surpasses C++ in Merge Sort across all dataset sizes and in Selection Sort and Insertion Sort for the majority of the datasets. This suggests Java's refined optimization for these algorithms, potentially due to its runtime environment and garbage collection mechanisms. Python's slower performance notes the impact of algorithm selection. Its interpreted nature results in longer execution times, but the relative scalability of Merge Sort and Quick Sort remains evident, highlighting their algorithmic efficiency.

This comprehensive examination of these findings in the figures illuminates the scale of performance differences between the programming languages, especially as the volume of data expands. The incremental increase in execution times highlights the importance of algorithm selection in the context of language capabilities and limitations. The data-based analysis provides valuable guidance in choosing sorting algorithms to optimize programming language efficiency. It further supports the importance of considering both the inherent capabilities of programming languages and the scalability of algorithms to achieve optimal performance for optimizing specific application requirements, software development objectives, and demands of data processing. Ultimately, these insights aid in navigating the complexities of computational efficiency. The proper combination of language and algorithm can significantly enhance the efficient handling of large datasets

Conclusion

In this study, we conducted a comprehensive comparative analysis of five sorting algorithms across three programming languages, highlighting major variations in efficiency and execution

time. Our findings reveal Quick Sort and Merge Sort as consistently outperforming others, reinforcing the effectiveness of divide-and-conquer strategies across different programming environments. Notably, Python's interpreted nature resulted in longer execution times compared to C++ and Java, emphasizing the impact of language choice on algorithm performance. This research illuminates the importance of selecting appropriate algorithms and programming languages for optimized data processing, offering valuable insights for software development and algorithm optimization. Future work may explore further experimentation with varied dataset sorting, datasets of sequentially increasing sizes, and the impact of new features in programming languages.

References

1. Al-Kharabsheh, K. S., AlTurani, I. M., AlTurani, A. M., Ibrahim, A. M., & Zanoon, N. I. (2013). Review on Sorting Algorithms: A Comparative Study. *International Journal of Computer Science and Security (IJCSS)*, 7(3), 124.

2. Durrani, S. A. K. N. (2015). Modified Quick Sort: Worst Case Made Best Case. *International Journal of Emerging Technology and Advanced Engineering*, 5(8).

3. Hayfron-Acquah, J. B., & D, P. (2015). Improved Selection Sort Algorithm. *International Journal of Computer Applications*, 110(5), 29–33.

4. Fangohr, H. (2004). A Comparison of C, MATLAB, and Python as Teaching Languages in Engineering. In *Computational Science - ICCS 2004* (pp. 1210–1217). Berlin, Heidelberg: Springer.