EUROPEAN JOURNAL OF SCIENTIFIC RESEARCH AND REVIEWS
2026; XX(XX): 1-9
10.5455/EJSRR.XXXXXXX

WisdomGale

ORIGINAL ARTICLE

Open Access

A controlled cross-language performance evaluation of merge sort under diverse data types and input patterns

Sydul Islam[1], Pranto Halder[2], Serazul Islam Saykat[3]

[1]Bachelor of Computer Science (Software Engineering), Faculty of Computing, Universiti Malaysia Pahang Al-Sultan, Pahang, Malaysia
[2]Department of Computer Science and Engineering, Faculty of Science and Engineering, Teesta University, Rangpur, Bangladesh
[3]Computer Science and Engineering, Faculty of Science and Engineering Teesta University, Rangpur, Bangladesh

ABSTRACT

Aim: This study evaluates the execution-time performance of iterative bottom-up Merge Sort across C, C++, Go, Java, PHP, and Python under controlled and identical algorithmic conditions. It investigates how programming-language design, runtime behavior, numeric data type, and input ordering influence practical performance despite Merge Sort maintaining O(nlogn) complexity across all cases.

Methods: Identical Merge Sort logic and identical input files were used for all six languages. Twelve experimental conditions were created from four numeric datasets—positive floating-point, negative floating-point, positive integer, and negative integer—and three input patterns: ascending, descending, and random. Each dataset contained 1,00,000 elements. Execution time was recorded in milliseconds, and overall mean, standard deviation, 95% confidence interval, and efficiency rank were calculated across the twelve conditions for each language.

Results: Go achieved the lowest overall mean execution time at 11.50 ms, followed by C at 19.83 ms, C++ at 22.75 ms, and Java at 23.58 ms. PHP and Python were substantially slower, with mean execution times of 156.33 ms and 556.75 ms, respectively. Random inputs generally produced slightly higher execution times, particularly in PHP and Python. The reported 0 ms C++ timings for floating-point datasets were interpreted as timer-resolution or compiler-optimization artifacts rather than literal zero execution time.

Conclusion: The findings show that programming-language execution models, memory representation, compiler behavior, and runtime overhead can substantially affect measured Merge Sort performance even when the algorithm and datasets are held constant. Lower-overhead compiled languages performed more efficiently in this experiment, while dynamically typed and interpreted environments incurred greater runtime costs. Because each condition was measured only once, the findings should be interpreted as descriptive benchmark results rather than definitive statistical evidence.

ARTICLE HISTORY

Received March 06, 2026
Accepted October 27, 2026
Published XX XXXX XXXX

KEYWORDS

Merge Sort; controlled benchmarking; programming languages; data types; input patterns; execution time; runtime overhead

Introduction

Sorting is a fundamental operation in computer science and underpins applications in databases, artificial intelligence, graphics, information retrieval, and data analytics. Efficient sorting improves data organization and often reduces the cost of downstream search and analysis. Merge Sort remains one of the most widely studied comparison-based algorithms because it offers stable performance and a time complexity of O(n log n) in the best, average, and worst cases [1,2]. Its divide-and-conquer structure also makes it attractive for external sorting, parallel processing, and workloads that require stability. Although its theoretical complexity is invariant to input order, measured runtime can still vary in practice due to hardware architecture, compiler behavior, runtime design, memory layout, and

Contact Pranto Halder ✉ prantohalder117@gmail.com Department of Computer Science and Engineering, Faculty of Science and Engineering, Teesta University, Rangpur-5404, Bangladesh.

© The Author(s) 2026. Open Access. This article is licensed under a Creative Commons Attribution 4.0 International License (CC BY). http://creativecommons.org/licenses/by/4.0/.

cache locality. For this reason, empirical evaluation remains necessary when the same algorithm is implemented across heterogeneous programming environments [3].

Modern software systems are built in languages with substantially different execution models, including ahead-of-time compiled languages such as C, C++, and Go and managed or interpreted environments such as Java, PHP, and Python. These differences can influence measured runtime through compiler optimizations, virtual-machine behavior, dynamic typing, object representation, and memory-management overhead [4,5]. Prior benchmarking work commonly reports that compiled languages often exhibit lower runtime overhead in CPU-intensive tasks [6]. However, practical runtime behavior is shaped not only by the language but also by the structure of the input data. Numeric representation (integer versus floating point, positive versus negative values) and input ordering (ascending, descending, and random) may alter memory-access patterns, branch predictability, and runtime overhead without changing Merge Sort's asymptotic complexity [7–9].

Existing studies on sorting performance often emphasize either theoretical analysis or single-language evaluation. For example, Sedgewick and Wayne [10] provide a foundational treatment of sorting algorithms, but not a controlled cross-language benchmark. Ayodele and Oluwade [6] compared several sorting algorithms in Java, C++, and Python and reported faster execution in compiled environments, yet their study did not isolate the effects of input type and input order under identical implementations. Related work on Merge Sort also tends to focus on parallelism or conceptual review rather than consistent multi-language experimentation [11,8].

Accordingly, the literature still provides limited controlled evidence on how the same Merge Sort logic behaves across multiple languages when both numeric representation and input ordering are varied systematically. The present study addresses that gap by evaluating six languages with identical algorithmic structure and standardized datasets, thereby focusing the analysis on observable runtime differences linked to language design, compiler behavior, and execution environment.

Against this backdrop, this paper presents a controlled cross-language evaluation of Merge Sort in C, C++, Go, Java, PHP, and Python. The goal is to examine how measured execution time changes when the same bottom-up Merge Sort logic is applied to datasets that differ in numeric type and input ordering. All experiments were conducted in a controlled laboratory environment to maximize comparability. The study is intended to provide practical evidence for developers, educators, and researchers who need to understand how language/runtime characteristics interact with algorithmic workloads in performance-sensitive contexts.

Section II offers the background of the study and its contextual background with the noticeable existing research. In section III, the methodology of the paper is described and the research design, data collection, and data analysis methods have been provided. Section IV will give the results and give interpretations and insights to the results. Section V discusses the research limitations and future direction of the research and gives possible places where research and practice can be taken.

Literature Review

Sorting algorithms have been studied extensively because of their central role in data organization, analysis, and system optimization. Merge Sort, in particular, is valued for its stable behavior and predictable O(n log n) complexity. While its theoretical properties are well established, practical performance still depends on contextual factors such as programming language, runtime system, data representation, and hardware environment.

A substantial body of work has examined sorting performance, but most studies remain limited in scope. Al-Raimi et al. [11] analyzed sequential and parallel Merge Sort in Java, emphasizing parallel speedup rather than cross-language comparison. Ayodele and Oluwade [6] compared Quick, Merge, and Insertion Sort in Java, C++, and Python, but did not study the role of input-type diversity under a controlled implementation framework. Ala'anzy et al. [8] surveyed sorting algorithms at a broader conceptual level, while Jilani and Sulaiman [9] highlighted that runtime systems such as garbage collection and virtual machines can influence measured performance. Classic texts by Knuth [12], Cormen et al. [2], and Sedgewick and Wayne [10] provide the theoretical foundation for algorithm analysis, yet they do not resolve how language/runtime effects manifest in a modern empirical benchmark. Likewise, Donovan and Kernighan [5] describe Go as a performant systems language, but do not present a controlled Merge Sort comparison across multiple languages.

From this body of literature, two major limitations emerge:

- Lack of comprehensive cross-language benchmarking using identical algorithmic logic. Most existing studies either test in a single language or use slightly different implementations, making comparisons less reliable.
- Insufficient focus on data type and input order diversity, which are critical in understanding performance in practical applications. Real-world data are often unstructured and varies in format (e.g., positive vs. negative values, float vs. integer), which may interact differently with each language's memory model and runtime environment.

A recurring limitation in the literature is the absence of a unified framework that evaluates Merge Sort under consistent algorithmic logic and identical dataset conditions. Prior work frequently varies the implementation strategy, the dataset construction process, or the programming environment, which makes cross-study comparisons difficult and sometimes misleading.

The present study responds to this limitation by applying the same bottom-up Merge Sort logic across six languages and by feeding each implementation the same dataset files in the same order. This design reduces implementation-related variability, although it does not eliminate all sources of measurement bias.

Justification for the present study

In light of these gaps, our study offers a controlled empirical investigation of Merge Sort performance across six widely used programming languages: C, C++, Go, Java, PHP, and Python. The benchmark varies four numeric datasets (positive and negative integers; positive and negative floating-point values) and three input orderings (ascending, descending, and random), producing 12 distinct test conditions for each language.

This approach enables us to:

- Measure how language design (compiled vs. interpreted) affects execution time in a controlled environment.
- Quantify the influence of data representation and structure on performance outcomes.
- Identify specific strengths and weaknesses of each language in handling algorithmically intensive tasks.

By combining language diversity, data variation, and a controlled experimental setup, this research contributes a clearer empirical view of how language/runtime choices shape measured algorithmic performance. The resulting observations are intended to support more careful language selection and more precise interpretation of algorithm benchmarks.

Materials and Methods

This section describes the experimental design used to compare Merge Sort performance across C, C++, Go, Java, PHP, and Python. The methodology was constructed to keep the algorithmic logic, the datasets, and the execution workflow as consistent as possible across languages so that measured differences would primarily reflect runtime behavior rather than major implementation changes.

A. Experimental System and Environment

All experiments were executed on a personal computer equipped with an AMD Ryzen 5 5500U processor (six physical cores, base clock 2.10 GHz) and 8 GB of RAM (5.86 GB usable). Because the dataset size fits comfortably in main memory, the benchmark reflects in-memory sorting rather than external sorting. Measured execution time is therefore influenced by CPU performance, memory hierarchy, compiler/runtime behavior, and cache locality rather than disk I/O.

To keep the workflow consistent, each program was run using the default toolchain available on the test machine and no additional optimization flags were enabled. The command sequence used was gcc MergeSort.c followed by ./a.exe for C, g++ MergeSort.cpp -o MergeSort followed by .\MergeSort.exe for C++, go run MergeSort.go for Go, java MergeSort.java for Java, php MergeSort.php for PHP, and python MergeSort.py for Python. Programs were executed while the system was idle to minimize background interference. We note, however, that running Go via go run may include build/startup overhead and that Java was invoked without a separate JVM warm-up phase; both choices can affect absolute timings and are therefore acknowledged as limitations of the benchmark design.

Dataset design and input categories

Each test dataset contained 1,00,000 elements. Four numeric categories were evaluated: positive floating-point values, negative floating-point values, positive integers, and negative integers. Positive and

negative cases were retained to capture possible differences in numeric representation and runtime handling across languages. Each dataset was then tested under three input orderings—ascending, descending, and random—yielding 12 total configurations per language. These input orderings were chosen to examine measured runtime under different data layouts; they do not change Merge Sort's asymptotic complexity, which remains O(n log n) regardless of input order. Any observed differences should therefore be interpreted as implementation- or runtime-level effects rather than as changes in the algorithm's theoretical complexity [13–15].

All input datasets were generated once and saved as plain-text (.txt) files so that every language implementation processed the same values in the same order. This identical-file workflow is central to the fairness of the study. The source code for the dataset generator and the six Merge Sort implementations is available in the project's public GitHub repository.

Experimental workflow overview

The entire workflow that was used in this study is visually represented in Fig. 1. This diagram shows a high-level view of the process that starts from dataset generation, moves through the preparation of Merge Sort implementations in each language, and ends with the data collection stages.

Each implementation read the same input file, executed the bottom-up Merge Sort procedure, and recorded elapsed execution time in milliseconds for the sorting task. Each of the 12 configurations was measured once. Because the benchmark did not repeat each configuration multiple times, the study does not support formal statistical testing of run-to-run variance. To provide a transparent summary of dispersion, the revised manuscript reports standard deviations and 95% confidence intervals computed across the 12 condition-specific execution times for each language. These statistics are descriptive across tested conditions rather than estimates of repetition-level measurement error.

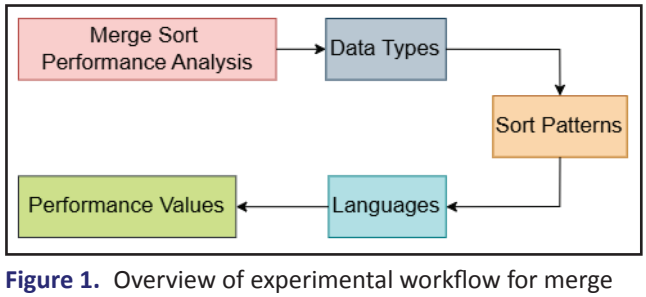


Figure 1. Overview of experimental workflow for merge sort performance analysis.

Merge sort implementation procedure

An iterative bottom-up version of Merge Sort was used in every language. This choice avoids deep recursive calls and keeps the control structure similar across implementations. Only syntax and language-specific data handling differ. Table 1 summarizes the shared algorithmic procedure used throughout the benchmark.

Table 1 demonstrates the algorithm whereby small blocks are used as the initial block, and the block size is doubled after every pass and merged. This is repeated until the entire array has been sorted. This is a stable behavior method, which is active in a variety of input patterns and data types. It also makes sure that the algorithm is consistent, simple, and can be recreated in any programming environment.

Execution time measurement and mean calculation

To quantify and compare performance, this study adopted a structured approach to execution time measurement. The sorting time for each data type and input order was recorded in milliseconds. Then, two key statistical measures were calculated to summarize the results.

The mean execution time for each data type i and language j was computed as:

$$\mu_{i,j} = \frac{1}{n}\sum_{k=1}^{n} t_{i,j,k}$$

where

- $t_{i,j,k}$ is the execution time for the k-th input order (Ascending, Descending, Random)
- $n = 3$ is the number of input orders

This calculation produced a mean value for each data type under each language, capturing the average behavior across different orderings [16].

Table 1. Procedure of the merge sort algorithm.

1	Start
2	Input array of size n
3	Initialize block size = 1
4	Repeat until block size ≥ n • Set left = 0 • For each pair of blocks: • Compute • mid = left + size – 1 • right = left + 2×size – 1 (bounded by n–1) • If mid < right, call Merge() • Update left = left + 2×size • Double block size (size = size × 2)
5	End when entire array is sorted
6	Output sorted array
7	Stop

Next, to assess overall language efficiency, the study averaged the four data-type-specific means for each language. The overall mean execution time for language j was determined using:

$$\overline{T_j} = \frac{1}{m}\sum_{i=1}^{m}\mu_{i,j}$$

where

- $m = 4$ is the number of Data Types

Based on $\overline{T_j}$, languages are ranked in terms of sorting efficiency. Lower values of $\overline{T_j}$ indicate better overall performance in executing the Merge Sort algorithm across diverse data conditions [17].

Overall, the methodology promotes fairness and comparability by maintaining identical datasets, closely matched algorithmic logic, and a controlled execution environment. At the same time, the revised analysis explicitly treats the results as descriptive empirical observations and avoids claims of formal statistical significance that would require repeated trials.

Table 2. Merge Sort execution times (ms) for float positive values by input order and language.

Data Description	C	C++	Go	Java	PHP	Python
Ascending	19	0	10	19	142	529
Descending	19	0	11	20	140	525
Random	19	0	11	24	180	614

Table 3. Merge Sort execution times (ms) for float negative values by input order and language.

Data Description	C	C++	Go	Java	PHP	Python
Ascending	19	0	11	18	137	526
Descending	19	0	11	19	146	524
Random	19	0	11	26	179	615

Table 4. Merge Sort execution times (ms) for integer positive values by input order and language.

Data Description	C	C++	Go	Java	PHP	Python
Ascending	19	42	10	22	136	535
Descending	19	41	11	24	141	529
Random	25	52	16	29	175	614

Table 5. Merge Sort execution times (ms) for integer negative values by input order and language.

Data Description	C	C++	Go	Java	PHP	Python
Ascending	18	43	11	28	174	524
Descending	18	43	10	23	141	527
Random	25	52	15	31	185	619

Results

This section presents the measured execution times of Merge Sort across the six languages, four numeric datasets, and three input orderings. Tables 2, 3, 4, 5 and 6 and Figs. 2, 3, 4, 5 and 6 summarize the condition-level results and the descriptive aggregate statistics.

Table 6. Overall language performance across all 12 test conditions (mean ± SD, 95% CI, and efficiency rank).

Language	Mean ± SD (ms)	95% CI (ms)	Rank
Go	11.50 ± 1.93	10.27–12.73	1
C	19.83 ± 2.44	18.28–21.39	2
C++	22.75 ± 24.01	7.50–38.00	3
Java	23.58 ± 4.25	20.88–26.29	4
PHP	156.33 ± 19.98	143.64–169.03	5
Python	556.75 ± 43.51	529.11–584.39	6

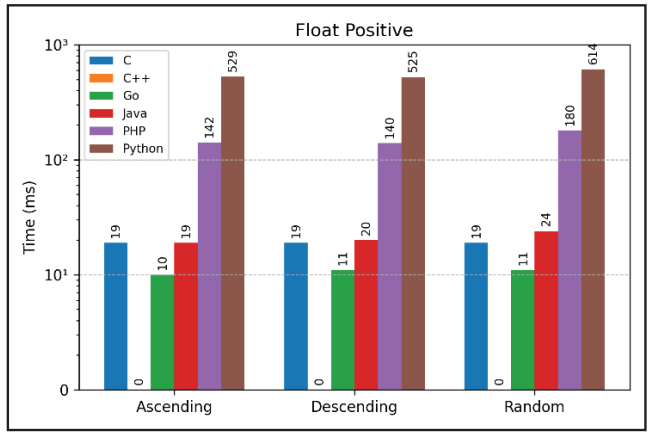


Figure 2. Execution time comparison for float positive data across ascending, descending, and random orders.

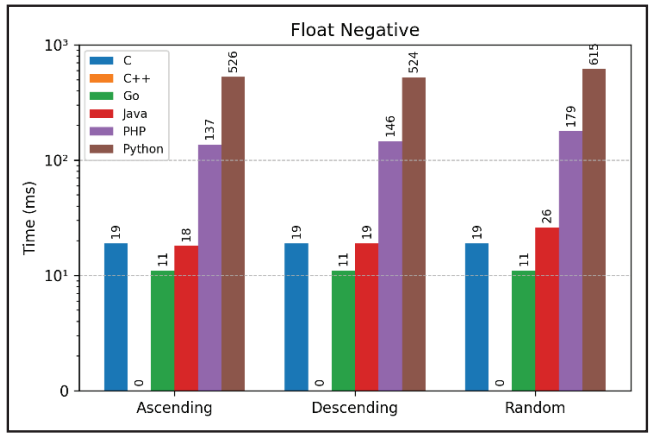


Figure 3. Execution time comparison for float negative data across ascending, descending, and random orders.

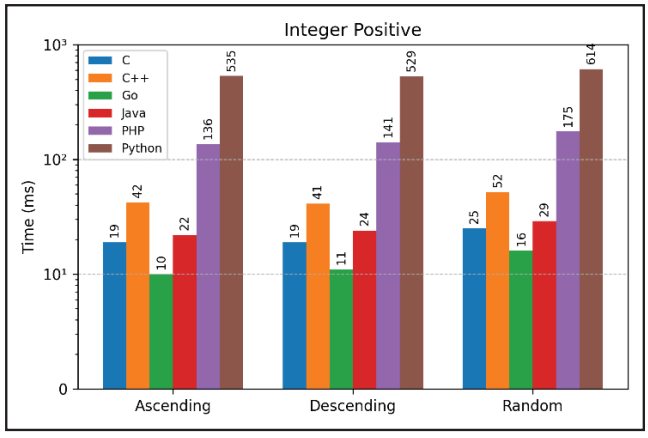


Figure 4. Execution time comparison for integer positive data across ascending, descending, and random orders.

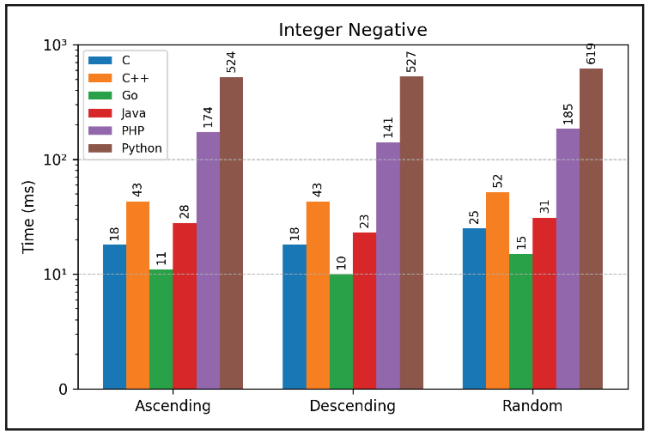


Figure 5. Execution time comparison for integer negative data across ascending, descending, and random orders.

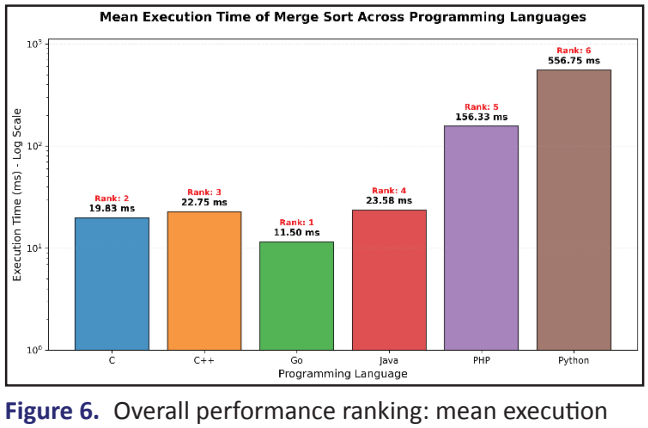


Figure 6. Overall performance ranking: mean execution times across all data types and input patterns.

Detailed runtime observations

Table 2 reports execution times for positive floating-point inputs. C remained completely stable at 19 ms across all three orderings. Go also showed consistently low times between 10 and 11 ms, while Java ranged from 19 to 24 ms. PHP required 142–180 ms and Python required 525–614 ms, making them the slowest environments in this condition set. The reported 0 ms values for C++ should not be read as literal zero-cost execution; they indicate that the observed runtime fell below the resolution of the timing mechanism or was affected by aggressive compiler optimization.

Table 3 shows a nearly identical pattern for negative floating-point inputs. C again remained fixed at 19 ms, Go stayed at 11 ms, and Java ranged from 18 to 26 ms. PHP and Python again incurred substantially higher costs, at 137–179 ms and 524–615 ms, respectively. The gap between Python and the compiled languages reflects measured runtime overhead rather than a change in Merge Sort complexity, which is unchanged across these conditions.

The anomalous 0 ms C++ timings in Tables 2 and 3 are best interpreted as a measurement artifact. In a millisecond-scale benchmark, very short runs can fall below timer resolution, and an optimization effect cannot be ruled out. The revised manuscript therefore treats these observations cautiously and does not attribute them solely to superior floating-point optimization.

Table 4 presents the results for positive integers. C again performed strongly at 19–25 ms, and Go remained the fastest language at 10–16 ms. Java occupied a moderate range of 22–29 ms. In contrast to the floating-point tables, C++ rose to 42–52 ms for integer inputs. That shift suggests that the earlier 0 ms float results should not be interpreted as a stable property of C++, but rather as an artifact requiring finer-grained profiling. PHP and Python again recorded the largest times, at 136–175 ms and 535–614 ms, respectively.

Table 5 contains the negative-integer results and shows a pattern similar to Table 4. C remained consistent at 18–25 ms, Go ranged from 10 to 15 ms, and Java ranged from 23 to 31 ms. C++ again required 43 ms for ascending and descending inputs and 52 ms for random input. PHP ranged from 141 to 185 ms, while Python recorded the highest times of the entire study, reaching 619 ms. These results reinforce the broad separation between the lower-overhead compiled languages and the higher-overhead managed or interpreted environments in this benchmark.

Comparison of language efficiency

Table 6 summarizes the aggregate language comparison using the 12 condition-specific execution times for each language. Go recorded the lowest overall mean execution time (11.50 ms), followed by C (19.83 ms), C++ (22.75 ms), and Java (23.58 ms).

PHP and Python were substantially slower, with overall means of 156.33 ms and 556.75 ms, respectively. Table 6 also reports the standard deviation and 95% confidence interval across the 12 tested conditions. These dispersion measures describe variability across conditions; they should not be interpreted as repetition-based uncertainty because each condition was executed once.

Figure 6 visualizes the same ranking pattern. Go clearly occupies the leading position, C follows at a modest distance, and C++, and Java remain in a similar middle-performance band. PHP and Python form a separate high-latency group. The wide gap between these groups indicates that language/runtime choice materially influences measured execution time even when the algorithmic logic is held constant.

Based on the observed performance ranges, the languages can be grouped descriptively into three tiers. The first tier contains Go and C, which combined low mean runtime with relatively small dispersion. The second tier contains C++ and Java, whose averages were still low but less stable, especially in C++ because of the float-timing anomaly. The third tier contains PHP and Python, both of which incurred substantially larger execution times across all tested conditions.

Impact of language characteristics

The measured differences can be interpreted in terms of runtime and memory behavior. Go's strong result is consistent with its ahead-of-time compilation, straightforward runtime, efficient handling of contiguous slices, and low-overhead implementation style for iterative loops. C also performed strongly because it adds very little runtime overhead beyond the compiled code itself. Java remained moderately competitive, likely because optimized array operations offset some of the costs associated with the JVM and automatic memory management, although the absence of a dedicated warm-up phase limits how precisely those timings can be interpreted. PHP and Python incurred the largest costs, which is consistent with the extra layers introduced by managed runtimes, dynamic typing, reference handling, and higher-level data abstractions.

Impact of input patterns and data types

Input ordering had only a small effect on the compiled languages in this benchmark. C and Go were nearly invariant across ascending, descending, and random inputs, while Java showed only modest increases for random orderings. In PHP and Python, however, random inputs produced noticeably larger measured times than ordered inputs. Because Merge Sort retains the same O(n log n) complexity across these cases, the observed differences are more plausibly associated with runtime overhead, memory-access behavior, and object handling than with a change in algorithmic work.

The effect of numeric representation was smaller than the effect of language/runtime choice, but it was not entirely absent. C, Go, and Java showed similar timings across floating point and integer datasets. C++ behaved differently between float and integer conditions, but that contrast is dominated by the sub-resolution float timings and should therefore be treated with caution. Overall, the benchmark suggests that numeric representation can influence measured runtime, yet the dominant source of variation remains the execution environment itself.

Taken together, the results indicate that lower-overhead compiled environments handled the benchmarked workloads more efficiently in this study, whereas higher-overhead runtimes amplified the cost of the same sorting logic, especially on randomly ordered data. These observations are descriptive rather than causal proof, but they are consistent with the broad runtime characteristics discussed in the literature [18–20].

Discussion

Despite the controlled design, the study has several limitations. First, all experiments were conducted on a single hardware platform, so the results may reflect processor-specific behavior, cache effects, and machine-level runtime interactions. Second, the benchmark evaluated only one algorithm (Merge Sort), so the conclusions should not be generalized automatically to other sorting methods.

Third, the benchmark used a single dataset size of 1,00,000 elements, which limits conclusions about scalability across very small or very large inputs. Fourth, each configuration was measured once, which prevents estimation of run-to-run variation within the same condition. Fifth, the protocol did not preserve detailed compiler/runtime version metadata or the exact timer APIs used in every language implementation, which reduces reproducibility. Finally, the study focused on execution time only; it did not measure memory consumption, garbage-collection activity, or energy use.

Future work should therefore expand the benchmark in several directions: repeated trials per condition, additional dataset sizes, more complete

environment metadata, and finer-grained profiling tools with sub-millisecond resolution. Additional sorting algorithms such as Quick Sort, Heap Sort, or Radix Sort could also be evaluated on multiple hardware platforms, including ARM-based and server-class systems. Incorporating memory, energy, and garbage-collection metrics would provide a more complete picture of language-level efficiency.

Conclusion

This study provides a controlled empirical comparison of Merge Sort across six programming languages under identical algorithmic conditions. The results show that runtime characteristics significantly influence measured execution time even when the algorithm itself is held constant. In this benchmark, Go achieved the lowest overall mean runtime, followed by C, while C++, and Java occupied a comparable middle range. PHP and Python were substantially slower across all tested conditions.

The revised analysis also clarifies two important interpretive points. First, input order did not create different theoretical cases for Merge Sort; the observed differences are practical runtime effects rather than changes in asymptotic complexity. Second, using common algorithmic logic and identical input files reduces implementation-related variability but does not eliminate all sources of bias. The anomalous 0 ms C++ float timings should therefore be viewed as measurement artifacts, not as direct evidence of zero-cost execution.

Within these limits, the benchmark still offers a useful descriptive snapshot of how language/runtime design shapes measured sorting performance in a controlled setting. The framework can be extended in future work to richer datasets, repeated trials, additional algorithms, and more detailed profiling, thereby providing a stronger basis for broader cross-language performance conclusions.

REFERENCES

1 von Neumann J. First Draft of a Report on the EDVAC. University of Pennsylvania; 1945.
2 Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to Algorithms. 3rd ed. Cambridge, MA: MIT Press; 2009.
3 Stroustrup B. The Design and Evolution of C++. Reading, MA: Addison-Wesley; 1994.
4 Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley; 1994.
5 Donovan AA, Kernighan BW. The Go Programming Language. Boston, MA: Addison-Wesley; 2015.
6 Ayodele OS, Oluwade B. A comparative analysis of Quick, Merge and Insertion sort algorithms using three programming languages I: execution time analysis. Afr J MIS. 2019;1(1):1–18.
7 Abdulkareem MB. Analysis of the performance of algorithms (K-Means, Farthest First, Hierarchical) using the data analysis and modeling tool Weka. Babylonian J Mach Learn. 2025:13–26.
8 Ala'anzy MA, Ramasamy KK, Bakar SA. Comparative analysis of sorting algorithms: a review. In: Proceedings of the 11th International Conference on Soft Computing and Machine Intelligence; 2024:1–6.
9 Jilani SA, Sulaiman S. Performance evaluation of sorting algorithms in different programming languages. J Comput Sci. 2013;9(1):10–16.
10 Sedgewick R, Wayne K. Algorithms. 4th ed. Boston, MA: Addison-Wesley; 2011.
11 Al-Raimi MS, Ali AY, Ramli R. Merge sort sequential and parallel programming: comparative analysis. Int J Comput Sci Inf Secur. 2024;22(3):1–8.
12 Knuth DE. The Art of Computer Programming. Volume 3: Sorting and Searching. 2nd ed. Boston, MA: Addison-Wesley; 1998.
13 Patel N, Banerjee A. Benchmarking sorting in JVM-based languages. Softw Pract Exp. 2022;52(3):421–438.
14 Green D, Lee M. Execution models and algorithm performance. ACM Comput Surv. 2023;55(1):1–38.
15 Kumar A. Real-time sorting in embedded systems: a case study. Microprocess Microsyst. 2019;71: 102894.
16 Torres F. Understanding cache behavior in sorting algorithms. Comput Archit Lett. 2021;20(2):98–101.
17 Omar M, Bhuiyan S. Sorting algorithm optimization with SIMD. J Parallel Distrib Comput. 2020;145:59–69.
18 Wong J, Lee S. Cache-aware sorting: analyzing locality of reference in Merge Sort. IEEE Trans Comput. 2020;69(8):1145–1158.
19 Vitek J, Zendra O. Memory management and garbage collection in programming language runtimes. ACM Comput Surv. 2019;52(2):1–35.
20 Viana ACT, Souza TBRS, Silva RGDAPL. Influence of compiler optimization levels on sorting algorithms performance. Inf Softw Technol. 2022;144:106806.
21 Okulicz K. Influence of programming languages on algorithm performance: a case study. Procedia Comput Sci. 2020;172:156–163.
22 Smith J, Zhao L. Evaluating sorting algorithms in cloud environments. IEEE Trans Cloud Comput. 2021;9(4):812–821.
23 Tanaka R. Memory allocation impact on sorting in interpreted languages. J Syst Archit. 2020;112:45–54.

24 Chen L. Sorting performance in multi-threaded Python and Go. Concurrency Comput Pract Exp. 2022;34(8):e6297.
25 Rahman A. An experimental study on the scalability of merge sort. Sci Program. 2021;2021:9937564.
26 Becker T. Algorithm complexity in managed languages. Inf Softw Technol. 2021;137:106602.
27 Lee J, Wong K. Energy profiling of sorting algorithms. Sustain Comput Inform Syst. 2022;35:100721.
28 Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to Algorithms. 3rd ed. Cambridge, MA: MIT Press; 2009.
29 Knuth DE. The Art of Computer Programming. Volume 3: Sorting and Searching. 2nd ed. Boston, MA: Addison-Wesley; 1998.
30 Diekmann D, Scholl A. Performance comparison of popular programming languages for matrix operations: a multi-language benchmark. Concurrency Comput Pract Exp. 2019;31(8).
31 Perri M, Verrastro F, Fiasche F. A comparative study of sorting algorithm performance in different programming languages (C, Java, Python). Int J Eng Adv Technol. 2023;12(4).
32 TIOBE Software. TIOBE Index. https://www.tiobe.com/tiobe-index/. Accessed April 2026.
33 Snyder KB. The effect of data type on sorting algorithm performance. J Comput Sci Technol. 2019;34(5):992–1002.