EUROPEAN JOURNAL OF SCIENTIFIC RESEARCH AND REVIEWS
2026; 4(1): 53-62
10.5455/EJSRR.20251025082011

RESEARCH ARTICLE

Open Access

Unifying algorithmic performance: Insertion sort across diverse programming paradigms

Md. Sydul Islam[1], Pranto Halder[2]

[1]Bachelor of Computer Science (Software Engineering), Faculty of Computing, Universiti Malaysia Pahang Al-Sultan, Pekan, Malaysia

[2]Computer Science and Engineering, Faculty of Science and Engineering, Teesta University, Rangpur, Bangladesh

ABSTRACT

Aim/Background: This study aims to systematically evaluate the practical runtime performance of the insertion sort algorithm across six widely used programming languages: C, C++, Go, Java, Hypertext Preprocessor (PHP), and Python. While insertion sort's theoretical time complexity is well established, real-world execution performance may differ significantly due to variations in language paradigms, compilation strategies, and runtime environments. The research focuses on identifying how language-specific characteristics influence actual execution time.

Methods: The insertion sort algorithm was implemented uniformly in all six languages using a shared pseudocode to ensure consistency. Experiments were conducted under controlled conditions using input sizes of 1 K, 10 K, and 100 K elements. Each dataset was tested with four different data arrangements: ascending, descending, nearly sorted, and random. Runtime measurements were collected and compared. Additionally, a Relative Execution Time Ratio (RETR) metric was introduced to provide a normalized and quantitative comparison of performance across languages.

Results: The experimental findings show clear performance differences among the programming languages. Go demonstrated the fastest runtimes across most scenarios, benefiting from its efficient compiled execution model. Java performed second-best, leveraging just-in-time compilation and runtime optimizations. C and C++ showed moderate performance, influenced by compiler optimization levels and memory management mechanisms. PHP and Python exhibited significantly slower execution times, particularly for large and poorly ordered datasets, mainly due to interpreter overhead and runtime inefficiencies. The RETR metric effectively highlighted these relative performance gaps.

Conclusion: The study confirms that programming language choice significantly impacts real-world algorithm performance, even for well-understood algorithms such as insertion sort. Compiled languages with efficient runtime optimizations generally outperform interpreted languages, especially for larger datasets. These results emphasize the importance of selecting appropriate programming languages for performance-critical applications and aligning algorithm implementation with both data characteristics and execution environments.

ARTICLE HISTORY

Received October 25, 2025
Revised February 14, 2026
Accepted February 14, 2026
Published February 27, 2026

KEYWORDS

Insertion sort; programming languages; runtime comparison; sorting algorithms; language efficiency

Introduction

Sorting algorithms are integral to the foundation of computer science and are extensively employed in a variety of computational domains, including database management, information retrieval, and numerical computation. Among these, the insertion sort algorithm is notable for its conceptual simplicity and educational value. Despite its quadratic time complexity of $O(n^2)$ in both average and worst-case scenarios, insertion sort remains relevant for small datasets and nearly sorted sequences, where it demonstrates relatively efficient performance with minimal overhead [1,2].

Contact Pranto Halder ✉ pranto@teestauniversity.ac.bd and prantohalder117@gmail.com Computer Science and Engineering, Faculty of Science and Engineering, Teesta University, Rangpur, 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/.

While the theoretical efficiency of an algorithm is crucial, practical performance often varies significantly depending on the programming language used for implementation. This variability arises due to differences in language design paradigms (e.g., compiled *vs.* interpreted), memory management models, runtime optimizations, and compiler efficiencies [3,4]. Consequently, the choice of programming language can influence the actual execution time of a given algorithm, even when its logical structure remains constant.

Numerous studies have compared programming languages using a variety of benchmarks to assess general performance, memory usage, and expressiveness [5–7]. However, there exists a relative paucity of focused, algorithm-specific performance studies that control for implementation uniformity across languages. Particularly, there is limited literature that investigates how the same fundamental algorithm behaves when implemented using consistent logic across multiple languages under identical runtime conditions.

To address this gap, the present study conducts a systematic performance evaluation of the insertion sort algorithm across six widely used programming languages: C, C++, Go, Java, Hypertext Preprocessor (PHP), and Python. The selection of languages spans compiled and interpreted paradigms, as well as both statically and dynamically typed languages, offering a broad perspective on language-influenced performance characteristics [8].

Each implementation adheres strictly to a shared pseudocode derived from canonical algorithmic definitions to ensure structural uniformity. The experiments are conducted within a standardized environment using Visual Studio Code's integrated terminal, on a consistent hardware configuration [9]. Execution times are recorded across datasets of varying sizes—1,000, 10,000, and 100,000 elements—and arrangements, including ascending, descending, nearly sorted, and random orders. This methodological rigor ensures that performance differences can be attributed to the languages themselves, rather than variations in algorithm design or testing conditions [10].

The results of this study offer insights into the practical implications of programming language selection for algorithmic tasks. By isolating and analyzing the influence of language-specific features on the runtime behavior of a well-understood sorting algorithm, this research contributes to the broader understanding of performance dynamics in modern programming environments.

Literature review

The comparative study of algorithm performance across programming languages has been a subject of ongoing research in computer science, particularly as language ecosystems continue to evolve with differing levels of abstraction, runtime environments, and performance characteristics.

Cormen et al. [1] and Sedgewick and Wayne [2] have provided foundational knowledge of sorting algorithms, including insertion sort, detailing their theoretical performance and practical applications. Insertion sort, despite its O(n^2) time complexity, remains a widely taught algorithm due to its simplicity and efficiency on small or nearly sorted datasets. Knuth [3] further elaborates on the adaptive nature of insertion sort, making it an appropriate choice for empirical performance evaluation when considering real-world data conditions.

While theoretical complexity offers insight into algorithmic efficiency, empirical studies are essential to understanding how these algorithms behave in practice, particularly across different programming languages. Prechelt [4] conducted one of the earliest notable empirical comparisons of programming languages, evaluating seven languages using a common problem set and metrics such as runtime, memory usage, and code length. His findings emphasized that language design and implementation have significant impacts on execution performance.

In more recent work, Hundt [5] compared loop recognition and optimization in languages such as C++, Java, Go, and Scala. His findings underscored the performance implications of language-level features, such as memory models and concurrency handling. Pike [6], in his discussion of the Go programming language, highlighted design choices made to balance readability, simplicity, and performance—attributes that potentially influence algorithm execution behavior.

Nanz and Furia [7] performed a large-scale comparative study using Rosetta Code, a platform that presents algorithmic solutions across multiple languages. Their analysis focused on maintainability, conciseness, and runtime, concluding that both high-level design and low-level execution models play roles in performance outcomes. Similarly, Le Goues and Weimer [8] emphasized the importance of measuring actual code quality and runtime behavior to guide language design improvements, reinforcing the need for targeted empirical studies.

Despite these contributions, limited research has focused explicitly on comparing insertion sort implementations across multiple languages using a consistent algorithmic structure and controlled testing conditions. Most existing studies evaluate complex or optimized algorithms, potentially masking the intrinsic performance effects introduced by the language itself. This paper aims to bridge that gap by applying a standardized, unoptimized insertion sort implementation across six languages—C, C++, Go, Java, PHP, and Python—to isolate and assess the direct influence of language characteristics on algorithmic runtime.

Materials and Methods

Figure 1 outlines the structured approach taken to evaluate the performance of the insertion sort algorithm across multiple programming languages. The process begins with the selection of insertion sort as the algorithm under study. Following this, the algorithm is implemented uniformly in six programming languages—C, C++, Go, Java, PHP, and Python—to ensure consistency in logic and structure. Concurrently, four types of input datasets are prepared: ascending, descending, nearly sorted, and random. These variations are used to assess how the algorithm performs under different data arrangements. The implementations are then subjected to a performance analysis, where execution times are recorded for each dataset-language combination. The results are systematically compared to identify performance trends and discrepancies across the languages. Finally, an evaluation is conducted to interpret the findings, drawing conclusions about the efficiency and behavior of insertion sort in different programming environments.

Environmental setup

To ensure fairness and consistency in performance comparison, a standardized insertion sort algorithm was implemented across all six programming.

The performance analysis of the insertion sort algorithm across six programming languages—C, C++, Go, Java, PHP, and Python—was conducted using a uniform experimental setup. All programs were executed on a system equipped with an AMD Ryzen™ 5 5500U processor with six cores operating at a base clock speed of 2.10 GHz, supported by 8 GB of RAM (5.86 GB usable) and a 512 GB SSD. The experiments were carried out using Visual Studio Code (version 1.96.2), with its integrated terminal serving as the execution environment for all the tests. This consistent setup ensured that the system's performance characteristics would not skew the results across the languages.

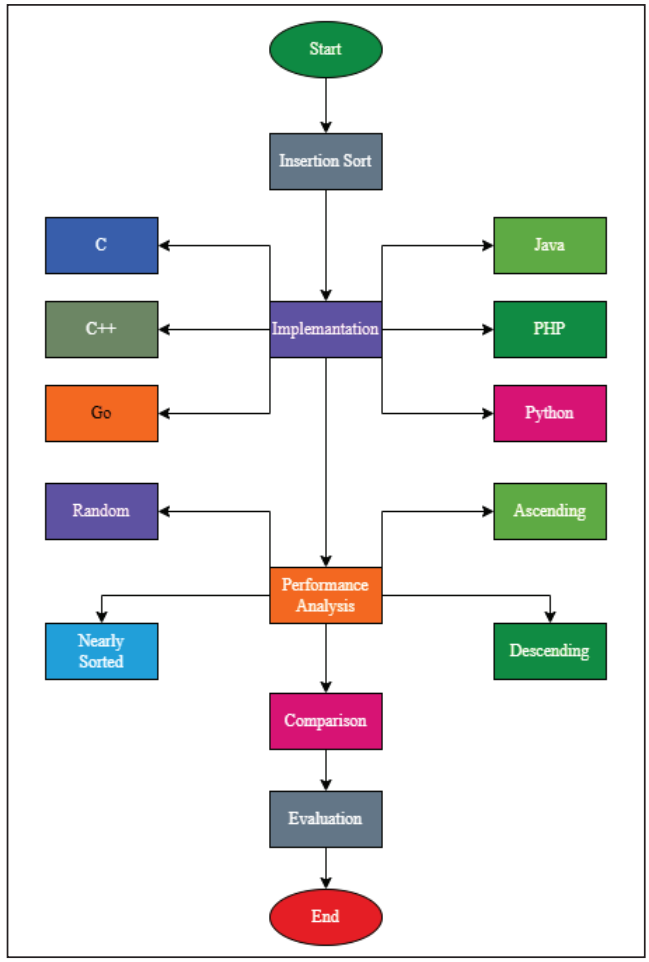


Figure 1. Experimental workflow: insertion sort performance analysis.

The experiments were executed sequentially from the same terminal on an otherwise idle system. To ensure a scientifically valid cross-language comparison, all timings are reported as wall-clock elapsed time measured using a monotonic high-resolution clock in each language. C uses clock_gettime(CLOCK_MONOTONIC) (or an equivalent monotonic timer), C++ uses std:chrono::steady_clock (or high_resolution_clock), Go uses time. Now/time. Java uses System. nanoTime, PHP uses hrtime(true), and Python uses time. perf_counter_ns. The timed region includes only the sorting routine; dataset generation, I/O, and printing are performed outside the measured interval. Because insertion sort is in-place, each trial sorts a fresh copy of the same input array.

For reproducibility, the compiler/interpreter versions and exact build commands should be recorded (e.g., via gcc --version, g++ --version, go

version, java -version, php -v, and python --version). All compiled implementations should be built with consistent optimization settings; for example, C: gcc –O3 –march = native, and C++: g++ –O3 –march = native -DNDEBUG. Interpreted/runtime languages are executed using their standard launchers (go run/go build, java -server, php, python).

To reduce measurement noise and enable statistical interpretation, each configuration (language, input size, and input pattern) should be repeated for multiple trials (e.g., 30). We recommend reporting the mean and standard deviation (and optionally the median) across trials. A brief warm-up run is performed before timing to stabilize caches; for Java specifically, a Java virtual machine (JVM) warm-up phase is required to allow just-in-time (JIT) compilation to optimize the hot loop (details provided in the following).

Java JIT warm-up: For each dataset configuration, the insertion sort function is executed K times (e.g., K = 20) as a warm-up within the same JVM process without recording the time. Timed trials are then performed for the next R runs using System. nanoTime around only the sorting call. To improve stability, the JVM is launched in server mode, and the heap size can be fixed (e.g., –Xms and –Xmx set to the same value) to reduce Garbage collection variance.

Example build/run commands (to be adapted to local paths):

C (gcc): gcc -O3 –march = native insertion.c -o insertion_c

C++ (g++): g++ -O3 –march = native -DNDEBUG insertion.cpp -o insertion_cpp

Go: go build -o insertion_go insertion.go

Java: javac InsertionSort.java; java -server InsertionSort

PHP: php insertion.php

Python: python insertion.py

External benchmarking and statistics: To obtain repeated-trial statistics in a uniform way across languages, an external harness such as hyperfine (or /usr/bin/time on Unix-like systems) can be used to measure wall-clock elapsed time. This approach applies the same timing method to every language and automatically reports mean, standard deviation, and run-to-run variance. For example, hyperfine can be invoked with warm-up runs and multiple measurements (e.g., hyperfine --warmup 5 --runs 30 ‘./insertion_c < input.txt’). Using an external harness also avoids accidentally timing dataset generation or printing, provided those are outside the program’s main timed path.

Pseudo code

To ensure fairness and consistency in performance comparison, a standardized insertion sort algorithm was implemented across all six programming languages. The algorithm’s structure was derived from its well-established pseudocode, which ensured that each implementation followed the same logical flow and operations. The pseudocode used as a reference is as follows [11].

```
1  for i = 1 to n-1 do
2     key = A[i]
3     j = i – 1
4     while (j >= 0) and (A[j] > key) do
5        A[j+1] = A[j]
6        j = j – 1
7     end while
8     A[j+1] = key
9  end for
```

Each implementation followed this structure without any optimizations or modifications specific to a language. For instance, in C and C++, loops and array indexing were employed to closely replicate the pseudocode [12], while Python utilized its native list indexing and loops to achieve the same functionality. Similarly, Java translated the pseudocode into its array-based syntax, Go leveraged its slice functionality for array manipulation, and PHP and Python maintained their dynamic array handling capabilities to mirror the logical flow [13].

This uniformity ensured that the differences in performance could be attributed solely to the underlying efficiency of the languages, their compilers, or interpreters, and not to any variations in the algorithm’s logic. By using a shared pseudocode foundation, the study eliminated biases stemming from implementation differences, which made the results both reliable and directly comparable across all languages [14]. This careful alignment of the insertion sort algorithm’s structure across languages underscores the rigor and objectivity of the experimental methodology [15].

Flowchart

Figure 2 illustrates the step-by-step logic of the insertion sort algorithm using standard flowchart symbols and a clear control structure. It begins

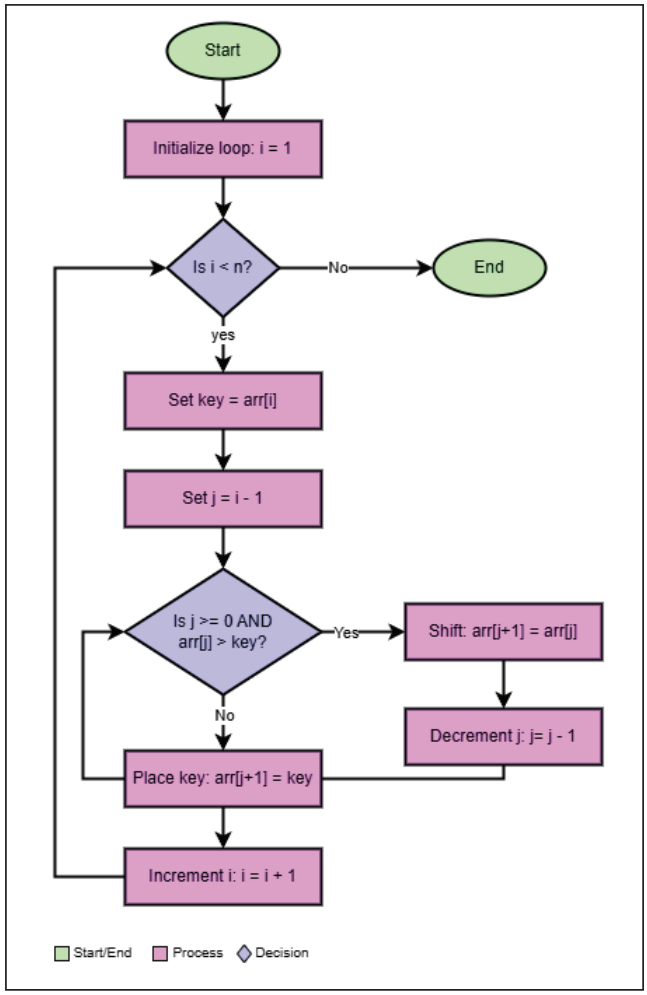


Figure 2. Algorithmic flowchart for insertion sort.

with the initialization of the outer loop at index i = 1, as the first element is considered sorted [16]. A decision is then made to check if i is less than the size of the array (n); if true, the algorithm proceeds to select the current element (arr[i]) as the key and sets j = i - 1. It then enters an inner loop that compares the key with elements to its left. If arr[j] is greater than the key, the value is shifted one position to the right (arr[j+1] = arr[j]), and j is decremented. This process repeats until the correct position for the key is found, at which point it is inserted (arr[j+1] = key) [17].

The outer loop then increments i and continues until all elements have been processed. The flowchart concludes with an end node once sorting is complete. This visual representation effectively captures the core logic and decision-making process of insertion sort, making it an excellent tool for understanding and teaching the algorithm [18].

The study involved running the insertion sort algorithm once for each language and dataset type. The datasets were categorized into ascending, descending, nearly sorted, and random arrangements to test the algorithm's performance under varying conditions. Each dataset type was further prepared in three sizes: 1,000 (1 k), 10,000 (10 k), and 100,000 (100 k) elements. This comprehensive dataset design provided a robust framework for evaluating how each language performs under different data arrangements and sizes.

To maintain uniformity during testing, the respective compilers and interpreters for each language were used via straightforward commands in the terminal. Specifically [19], gcc for C, g++ for C++, Go's native compiler go for Go, the java command for Java, the php interpreter for PHP, and the python interpreter for Python. These commands followed standard conventions and ensured optimal code execution for each language [20].

To support reproducibility and enable future comparisons, this study provides open access to all source code used in the experiments. The repository includes insertion sort implementations in C, C++, Go, Java, PHP, and Python. Researchers and developers can review or reuse the code by visiting the following GitHub link: https://github.com/mdsydulislam/InsertionSortPerformanceAnalysis.git

Relative execution time ratio (RETR)

To quantitatively compare runtime efficiency across programming languages, we define the RETR, a simple normalized ratio between two languages under the same input size and pattern.

For languages L1 and L2, dataset size n, and input pattern p, RETR is defined as: RETR_{L1→L2}(n,p) = T_{L1}(n,p) / T_{L2}(n,p), where T_{L}(n,p) is the measured execution time of language L for that configuration.

$$\mathrm{RETR}_{L_1 \to L_2}^{(S,P)} = \frac{T_{L_1}^{(S,P)}}{T_{L_2}^{(S,P)}}$$

Where T_{L1}(n,p) and T_{L2}(n,p) are the measured wall-clock execution times (in milliseconds) for languages L1 and L2 under the same configuration.

- Interpretation: RETR_{L1→L2} > 1 means L1 is slower than L2 by that factor; RETR_{L1→L2} < 1 means L1 is faster than L2; and RETR_{L1→L2} = 1 indicates equal performance.
- For example, RETR_{C→Java}(100 k, Descending) = X means that C is X× slower than

Java for sorting 100,000 elements in descending order (equivalently, Java is X× faster than C for that configuration).

RETR complements the absolute timing tables by making cross-language comparisons interpretable even when execution models differ (compiled *vs.* interpreted, ahead-of-time *vs.* JIT). In this paper, we use RETR to compare language pairs under the same workload conditions.

This methodology ensures the consistency and reliability of the performance comparisons while providing a realistic assessment of the insertion sort algorithm's behavior across different programming languages. By capturing the impact of dataset type, size, and language implementation in a systematic manner, the study offers valuable insights into the interplay between algorithmic efficiency and programming language performance in real-world scenarios.

Results

This section presents and critically examines the empirical performance of the insertion sort algorithm implemented across six programming languages—C, C++, Go, Java, PHP, and Python—on datasets of increasing size (1,000, 10,000, and 100,000 elements) and varying data distributions (ascending, descending, nearly sorted, and random). The execution time, measured in milliseconds, was recorded for each configuration, and the results are visualized in Figures 3–5, supported by Tables 1–3.

Performance trends across input sizes

The results from the experimental evaluation illustrate a clear and consistent relationship between input size and execution time for insertion sort across all tested programming languages and input configurations. As shown in Figure 3 (1,000 elements), Figure 4 (10,000 elements), and Figure 5 (100,000 elements), execution time increases non-linearly with input size. This trend becomes more obvious after the input size crosses 10,000 elements. It confirms the effect of insertion sort's time complexity, O (n^2), in real-world scenarios [7,21].

Table 1 gives the execution times for 1,000 elements. At this size, most programming languages show only small differences—usually less than 100 milliseconds—except under worst-case conditions. However, in Table 3, which covers 100,000 elements, the differences between languages grow much larger. For instance, C takes 4,665 milliseconds to sort random data, while Python takes 368,869 milliseconds for the same task. That is about 79 times slower. This result shows that input size becomes a much more serious factor in performance when working with large datasets [22].

The sharp rise in execution time from 10,000 to 100,000 elements, particularly for slower languages (e.g., Python and PHP), is not merely due to algorithmic complexity, but also to factors such as

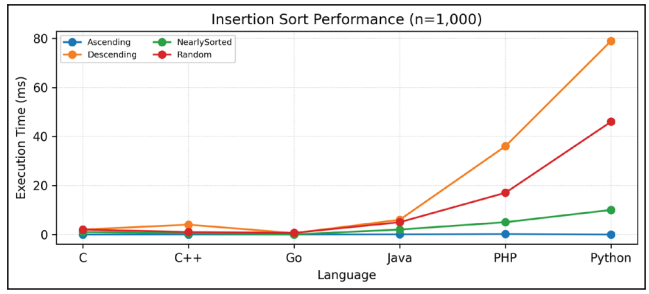


Figure 3. Insertion sort performance for 1,000 elements.

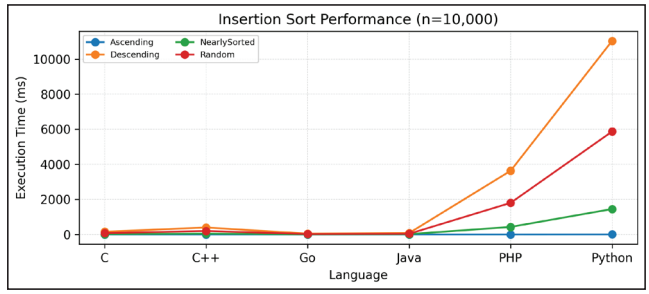


Figure 4. Insertion sort performance for 10,000 elements.

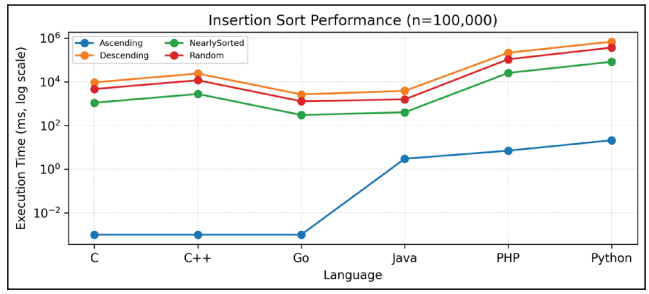


Figure 5. Insertion sort performance for 100,000 elements (log-scale y-axis; values below 0.001 ms are clipped).

Table 1. Execution times (1,000 Elements).

Language	C	C++	Go	Java	PHP	Python
Ascending	0	0.0077	0	0.0541	0.1609	0
Descending	2	4	0.5389	6	36	79
NearlySorted	1	0.4535	0	2	5	10
Random	2	1	0.6336	5	17	46

Table 2. Execution times (10,000 Elements).

Language	C	C++	Go	Java	PHP	Python
Ascending	0	0.0743	0	0.7797	1	3
Descending	155	397	45	79	3631	11041
NearlySorted	18	48	6	17	429	1449
Random	77	198	23	37	1801	5871

Table 3. Execution times (100,000 Elements).

Language	C	C++	Go	Java	PHP	Python
Ascending	0	0.0001	0	3	7	21
Descending	9477	23872	2650	3832	215052	685860
NearlySorted	1095	2783	302	401	25492	82528
Random	4665	11799	1284	1571	107608	368869

memory access patterns, language runtime overhead, and compiler optimizations. These technical factors increase the performance gap as the input size grows.

Impact of initial data arrangement

The way data is arranged before sorting has a strong effect on how well insertion sort performs. Figures 3–5 and Tables 1–3 clearly show this impact. In all input sizes and programming languages, data that is already in ascending order produced the fastest results. This follows the best-case time complexity of insertion sort, which is O (n) [22]. In this case, the algorithm only compares values without moving any elements. Even with 100,000 elements, languages such as Go, Java, and C finished sorting in less than 10 milliseconds, and in some cases, even faster than 1 millisecond (Table 3). These results show that insertion sort works very efficiently when data is nearly or fully sorted [23].

On the other hand, data in descending order caused much slower performance, reflecting the worst-case behavior of insertion sort. Figure 5 (log-scale y-axis) shows that Python required 685,860 ms (≈686 s, ≈11.4 min) to sort 100,000 elements in descending order. In comparison, it took only 21 ms for the same number of elements in ascending order—an increase of over 31,000 ×. This gap illustrates that input structure can influence runtime as strongly as input size: in the worst case, each new key must be shifted past all previously sorted elements, sharply increasing the number of operations [24].

Data that is nearly sorted gave much better results than random or descending data. This matches the nature of insertion sort. When elements are close to their correct positions, fewer moves are needed. In all programming languages tested, nearly sorted input gave times much closer to ascending data than to random or descending data. This trend is visible in Figures 4 and 5 [25]. It confirms that insertion sort is well-suited for situations where the input is almost sorted [26].

Programming language performance comparison

The data strongly supports the conclusion that programming language implementation has a profound impact on the runtime efficiency of insertion sort, especially at scale. As highlighted in Table 3, Go consistently outperformed all other languages across every input size and configuration. It runs faster because it is compiled [27]. It runs close to the machine level, uses memory more efficiently, and has very little runtime overhead.

For instance, when sorting descending data with 100,000 elements, Go took 2,650 milliseconds, while other languages took more than twice as long—even C++ exceeded 23,000 milliseconds (Table 3). These results highlight that when performance is a critical requirement, especially for large datasets, compiled languages offer a significant advantage [28].

Conversely, Python emerged as the slowest performer. As an interpreted language, Python adds a lot of runtime overhead. This led to very long execution times, such as 368,869 milliseconds for 100,000 random elements and 685,860 milliseconds for descending input (Table 3). PHP also exhibited similar limitations but maintained a consistent performance edge over Python [2,10,28].

Compiled and JIT-managed languages (C, C++, Go, and Java) execute the same O(n^2) insertion sort inner loop, so relative performance differences are mainly driven by code generation quality, bounds-check elimination, and the efficiency of the memory-write pattern in the worst-case (descending) input. For n = 100,000 in descending order, insertion sort performs approximately $n(n-1)/2$ element shifts, making the runtime largely memory-write dominated; small per-iteration overheads can therefore amplify into large end-to-end differences.

The large C versus C++ gap observed in the descending case is not expected when both use a contiguous integer array and are compiled with the same optimization level. In practice, a multi-x slowdown often indicates an implementation or build mismatch (e.g., bounds-checked access via vector::at, extra copying of the array within the timed region, use of std::swap in the inner loop instead of shift-with-key, debug builds, or I/O/printing affecting the measurement). For a fair comparison, both

implementations should use the same data structure (raw array or std::vector with operator[]), the same loop structure, and identical optimization flags (e.g., -O3 –march = native -DNDEBUG).

Go's advantage can be explained by its optimizing compiler producing a tight loop over contiguous slices, often eliminating bounds checks when the loop structure is simple, and by generating efficient machine code via its SSA backend. Java can similarly approach native performance after JIT warm-up because HotSpot may inline the hot loop and eliminate array bounds checks. Nevertheless, Go outperforming optimized C is atypical for a write-intensive O(n^2) loop, so the exact magnitude of the gap should be interpreted only after harmonizing timers to wall-clock time and aligning build flags and timed regions across languages.

Performance scaling analysis

The way each programming language handles larger input sizes gives important information about its efficiency and usefulness in real-world applications. When sorting 1,000 elements (Fig. 3), all languages performed well [29]. Even Python completed most tasks in under one second. But at 10,000 elements (Fig. 4), slower languages began to show clear weaknesses. The increase in execution time became more noticeable, especially in interpreted languages like Python and PHP [30].

When the input size reached 100,000 elements (Fig. 5), the differences between languages became much larger. For example, Python's descending-case time grew from 79 ms at n = 1,000 (Fig. 3) to 685,860 ms at n = 100,000 (Fig. 5), i.e., ≈8,680 × [31]. This growth reflects insertion sort's $O(n^2)$ behavior amplified by higher runtime overhead in interpreted environments. In contrast, the compiled/JIT languages show smaller proportional increases because their per-iteration overhead is lower [32].

Performance consistency across data types

When examining consistency across different data types, Go and Java exhibited stable and predictable performance patterns. They kept similar performance patterns between input types at all dataset sizes [33]. This level of consistency shows that both languages can provide reliable and stable behavior, which is important for systems that require dependable performance [34].

Python and PHP, on the other hand, displayed irregularities in performance consistency. The performance gaps between ascending and descending inputs were vastly larger than those observed in other languages. This large difference may show that their standard sorting methods or the way they manage data are not well optimized for insertion sort [35].

C and C++ fell into a middle category, delivering moderate performance depending on the data pattern. While C performed well overall, it showed notable strength and outperformed C++ by a wide margin [36].

Discussion

Implications for algorithm and language selection

These findings carry important implications for developers choosing sorting algorithms and programming languages in practice. For small datasets (≤1,000 elements), the difference in execution times is negligible across all languages. In this case, insertion sort works well and remains a simple and effective choice, no matter which language is used [35,36]. This is reinforced by the minimal differences observed in Table 1 and Figure 3.

However, for large datasets, especially with non-ideal input configurations (random or descending), both algorithmic efficiency and language execution model become crucial. Insertion sort has a time complexity of O (n^2) [37], which leads to very long runtimes in slower languages. This makes it a poor choice for large datasets unless the data is nearly sorted.

Developers working on performance-sensitive applications would benefit from avoiding interpreted languages for sorting large datasets unless high-level algorithm replacements or native extensions are used. Compiled languages such as C and Go not only maintain excellent performance but also show robustness across different data arrangements. This makes them better suited for systems that deal with large amounts of data.

In conclusion, while insertion sort remains effective in scenarios involving small or nearly sorted datasets, its practicality diminishes quickly as data volume increases—particularly in higher-overhead languages. Therefore, careful consideration of both data characteristics and programming language should guide the selection of sorting techniques in software systems.

Conclusion

This study offers an empirical view of how language execution models and runtime designs affect the observed execution time of the same insertion sort logic. The results reinforce that asymptotic complexity alone is not sufficient to predict practical performance: constant factors from compilation strategy (ahead-of-time *vs.* JIT), bounds checking, and runtime overhead can meaningfully shape outcomes.

From an engineering perspective, insertion sort remains a reasonable choice for small inputs or nearly sorted data, where its low overhead and cache-friendly behavior can be advantageous. For large inputs with random or descending order, the quadratic number of element shifts dominates, and the algorithm becomes impractical in higher-overhead runtimes; production systems should prefer O(n log n) sorting algorithms or hybrid strategies (e.g., insertion sort as a base case within introsort/timsort).

Methodologically, rigorous cross-language benchmarking requires harmonized wall-clock time measurement, an explicit warm-up procedure for JIT-based languages, and repeated trials with reported variance. The proposed RETR complements absolute timings by expressing relative slowdowns between languages and input patterns in an interpretable way.

Overall, the study highlights that those apparent differences in "language speed" are often a combined effect of the compiler/runtime, build flags, and measurement protocol—not solely the language itself. Clear reporting of these factors improves reproducibility and helps developers make more reliable performance-sensitive decisions.

Limitations and future directions

Limitations: This study benchmarks insertion sort on integer arrays using a single hardware/OS configuration, so absolute times may vary across CPUs, operating systems, compiler versions, and runtime settings. Because cross-language microbenchmarks are sensitive to timing functions, warm-up effects (for JIT runtimes), and OS scheduling, the benchmarking protocol should use monotonic wall-clock timers, a Java warm-up phase, and repeated trials with variance reporting to support statistically reliable claims.

Future work will extend the evaluation to additional sorting algorithms (e.g., quicksort, mergesort, timsort/introsort hybrids), broader data types, and multiple hardware platforms, and will incorporate deeper profiling (cache misses, branch behavior, and memory bandwidth) and energy measurements to explain performance differences beyond end-to-end time.

References

1. Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to Algorithms. 3rd edition, MIT Press, Cambridge, MA, 2009.
2. Sedgewick R, Wayne K. Algorithms. 4th edition, Addison-Wesley, Boston, MA, MA, 2011.
3. Knuth DE. The Art of Computer Programming. Vol 3: sorting and Searching. 2nd edition, Addison-Wesley, Boston, MA, 1998.
4. Prechelt L. An empirical comparison of seven programming languages. Computer 2000; 33(10):23–9.
5. Hundt R. Loop recognition in c++/java/go/scala. Proceedings of Scala Days, 2011, 2011(86), p. 2.
6. Pike R. Go at Google: language design in the service of software engineering. Commun ACM 2012; 54(2):58–65.
7. Nanz S, Furia CA. A comparative study of programming languages in Rosetta Code. In Proceedings of the 37th International Conference on Software Engineering (ICSE), 2015, 2015, Vol. 1, pp. 778–88.
8. Le Goues C, Weimer W. Measuring code quality to improve language design. IEEE Softw 2012; 29(6):44–51.
9. Durrani OK, Farooqi AS, Chinmai AG, Prasad KS. Performances of sorting algorithms in popular programming languages. In 2022 International Conference on Smart Generation Computing, Communication and Networking (SMART GENCON), IEEE, 2022, pp. 1–7.
10. Rizvi QM, Rai H, Jaiswal R. Sorting algorithms in focus: a critical examination of sorting algorithm performance. In Emerging trends in IoT and computing technologies CRC Press, 2024, pp. 103–6.
11. Fagbola TM, Thakur SC. Investigating the effect of implementation languages and large problem sizes on the tractability and efficiency of sorting algorithms. Int J Eng Res Technol 2019; 12(2):196–203.
12. Durrani, O.K. and Abdulhayan, S. November. Performance measurement of popular sorting algorithms implemented using Java and Python. In 2022 International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME), IEEE, 2022, pp. 1–6.
13. Prajapati P, Bhatt N, Bhatt N. Performance comparison of different sorting algorithms. Int J Latest Technol Eng Manag Appl Sci 2017; 6(6):39–41.
14. Chhatwani PK, Somani JS. Comparative analysis and performance of different sorting algorithm in

data structure. Int J Adv Res Comput Sci Softw Eng 2013; 3(11):500–7.
15. Gul M, Amin N, Suliman M. An analytical comparison of different sorting algorithms in data structure. Int J Adv Res Comput Sci Softw Eng 2015; 5(5):1289–98.
16. Al-Amin IM, Okeyinka AE, Ibrahim A. Comparative complexity study of bubble sort and insertion sort using Java programming language: a review. J Sci Innov Technol Res 2024; 3(9). Available via https://africanscholarpub.com/ajsitr/article/view/141
17. Akhter N, Idrees M, Rehman F. Sorting algorithms—a comparative study. Int J Comput Sci Inf Secur 2016; 14(12):930–5.
18. Al-Kharabsheh KS, AlTurani IM, AlTurani AMI, Zanoon NI. Review on sorting algorithms: a comparative study. Int J Comput Sci Secur 2013; 7(3):120–6.
19. Goel K, Dwivedi P, Sharma O. Performance analysis of various sorting algorithms: Comparison and optimization. In 2023 11th International Conference on Intelligent Systems and Embedded Design (ISED), IEEE, 2023, pp. 1–5.
20. Marcellino M, Pratama DW, Suntiarko SS, Margi K. Comparative of advanced sorting algorithms (quick sort, heap sort, merge sort, intro sort, radix sort) based on time and memory usage. In 2021 1st International Conference on Computer Science and Artificial Intelligence (ICCSAI), IEEE, 2021, 2021, Vol. 1, pp. 154–60.
21. Enoch O, Joseph O, Ok E, Williams B, Aria J, Jose D, et al. Data Structures and Algorithms: Trees, graphs, and hash tables, Sorting and searching algorithms. [online] ResearchGate 2024. Available via https://www.researchgate.net/publication/387238508 (Accessed 21 February 2025).
22. Aly AG, Jensen AE ElAarag H. Improving merge sort and quick sort performance by utilizing Alphadev's sorting networks as base cases. In Proceedings of the 2025 ACM Southeast Conference, 2025, pp. 239–44.
23. Li X, Garzaran MJ, Padua D. Optimizing sorting with genetic algorithms. In International Symposium on Code Generation and Optimization, IEEE, 2005, pp. 99–110.
24. Li X, Garzaran MJ, Padua D. Optimizing sorting with machine learning algorithms. In IEEE International Parallel and Distributed Processing Symposium, IEEE, 2007, 1–6.
25. Xingyi Zhang, Ye Tian, Ran Cheng, Yaochu Jin. An efficient approach to nondominated sorting for evolutionary multiobjective optimization. IEEE Trans Evol Comput 2014; 19(2):201–13.
26. Ala'Anzy MA, Mazhit Z, Ala'Anzy AF, Algarni A, Akhmedov R, Bauyrzhan A. Comparative analysis of sorting algorithms: a review. In 2024 11th International Conference on Soft Computing & Machine Intelligence (ISCMI), IEEE, 2024, pp. 88–100.
27. Deb K, Agrawal S, Pratap A, Meyarivan T. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In International conference on parallel problem solving from nature, Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 849–58.
28. Musser DR. Introspective sorting and selection algorithms. Softw Pract Exp 1997; 27(8):983–93.
29. Jain H, Deb K. An evolutionary many-objective optimization algorithm using reference-point based nondominated sorting approach, part II: handling constraints and extending to an adaptive approach. IEEE Trans Evol Comput 2013; 18(4):602–22.
30. Li X. A non-dominated sorting particle swarm optimizer for multiobjective optimization. In Genetic and evolutionary computation conference. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 37–48, 2003.
31. Mahmoud HM. Sorting: A Distribution Theory. John Wiley & Sons, New York, NY, 2000.
32. Kuriakose S, Shunmugam MS. Multi-objective optimization of wire-electro discharge machining process by non-dominated sorting genetic algorithm. J Mater Process Technol 2005; 170(1-2):133–41.
33. Mehlhorn K. Data structures and algorithms 1: Sorting and searching. Springer Science & Business Media, 2013.
34. Mandal D, Pal SK, Saha P. Modeling of electrical discharge machining process using back propagation neural network and multi-objective optimization using non-dominating sorting genetic algorithm-II. J Mater Process Technol 2007; 186(1-3):154–62.
35. Greb A, Zachmann G. GPU-ABiSort: optimal parallel sorting on stream architectures. In Proceedings 20th IEEE International Parallel & Distributed Processing Symposium, IEEE, 2006, pp. 10–pp.
36. Premkumar M, Jangir P, Sowmya R, Alhelou HH, Heidari AA, Chen H. MOSMA: multi-objective slime mould algorithm based on elitist non-dominated sorting. IEEE Access 2020; 9:3229–48.
37. Cerf S, Doerr B, Hebras B, Kahane Y, Wietheger S. The first proven performance guarantees for the Non-Dominated Sorting Genetic Algorithm II (NSGA-II) on a combinatorial optimization problem. 2023. arXiv preprint arXiv:2305.13459.