EUROPEAN JOURNAL OF SCIENTIFIC RESEARCH AND REVIEWS
2026; 4(1): 14-28
10.5455/EJSRR.20251025063150

WisdomGale

RESEARCH ARTICLE

Open Access

Language aware sorting: Empirical and machine learning driven performance evaluation of insertion sort

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

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

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

ABSTRACT

Aim/Background: Insertion Sort is widely taught in computer science, yet its practical runtime can vary substantially depending on the programming language and the structure of the input data. This study aims to provide a language-aware empirical comparison of Insertion Sort and to complement benchmark results with machine-learning–based runtime classification.

Methods: Insertion Sort was implemented and evaluated in six languages (C, C++, Go, Java, PHP, and Python). Execution time was measured under a controlled hardware/software environment using identical datasets of three sizes (1,000, 10,000, and 100,000 elements) and four input patterns (ascending, descending, nearly sorted, and random). To interpret performance trends, supervised machine learning models were trained to classify runtime behavior into Fast, Moderate, and Slow categories, and a Relative Execution Time Ratio (RETR) metric was introduced to normalize runtimes against the fastest observed case.

Results: Go achieved the best overall performance, followed by Java, while C and C++ produced mid-range results. Python and PHP were consistently the slowest, with the largest slowdowns observed for large and reverse-ordered inputs. For runtime classification, Random Forest outperformed XGBoost, achieving 93% accuracy compared with 87% for XGBoost. RETR enabled a clearer cross-language comparison by standardizing runtimes relative to the fastest baseline.

Conclusion: Insertion Sort performance is strongly influenced by both programming language and input characteristics. Combining benchmarking with machine learning based classification and normalized metrics (RETR) offers a more interpretable view of runtime behavior and supports better alignment of algorithm choices with language and data context in real-world applications.

ARTICLE HISTORY

Received October 25, 2025
Accepted December 20, 2025
Published February 04, 2026

KEYWORDS

Insertion sort; programming languages; runtime performance; data patterns; Machine Learning; random forest; XGBoost; relative execution time ratio (RETR)

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 clear logical structure that facilitates understanding for beginners. Despite its quadratic time complexity of $O(n^2)$, which can lead to inefficiencies with larger datasets, Insertion Sort is particularly effective for small datasets and datasets that are nearly sorted, where it performs optimally [1,2]. 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 significantly affect execution time due to differences in compilation, memory management, and

Contact Pranto Halder ✉ 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/.

runtime environments, 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 exist limited studies specifically examining the performance of algorithms across different programming languages that control for implementation uniformity across languages. Particularly, only a small number of studies have systematically examined how the same sorting algorithm behaves across multiple languages while controlling implementation structure, dataset characteristics, and runtime environments [8]. 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, 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 [9]. 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 [10]. 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 [11].

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 [12].

This research paper is systematically structured to present a comprehensive investigation of Insertion Sort performance across multiple programming languages. The study begins with a thorough literature review in Section II, critically analyzing previous work on sorting algorithms and language performance while identifying key research gaps. Section III details a rigorous methodology based on standardized Insertion Sort implementations across six programming languages and carefully structured datasets of three sizes and four input patterns, with execution times recorded and compared using the Relative Execution Time Ratio (RETR) metric for performance evaluation, alongside a machine learning–based classification approach that predicts performance categories using models such as Random Forest and XGBoost. Section IV presents and discusses the empirical results, featuring comparative language performance analysis, scaling effects, and machine learning classification outcomes. Finally, Section V examines the study's limitations and proposes valuable directions for future research, such as investigating additional algorithms and optimization techniques. This logical progression from theoretical foundations through methodological implementation to empirical validation and future considerations ensures a robust and complete research contribution to the field of algorithm performance analysis. The overall organization of the research is outlined in Figure 1, which provides a structural roadmap from literature synthesis to experimental validation and conclusion.

Literature Review

The performance of sorting algorithms across different programming languages has been a significant area of research in computer science, with implications for improving algorithm optimization and execution efficiency. Numerous studies have explored algorithmic efficiency, language-specific execution behaviors, and performance comparisons across widely used programming languages such as C, C++, Java, Python, and Go. These investigations commonly focus on execution time, scalability, and memory utilization. This section synthesizes prior research relevant to the empirical evaluation of Insertion Sort and its performance variations across programming languages. Several comparative studies and reviews evaluate classical sorting algorithms and their practical behavior across contexts [13,14,17,18,21,22].

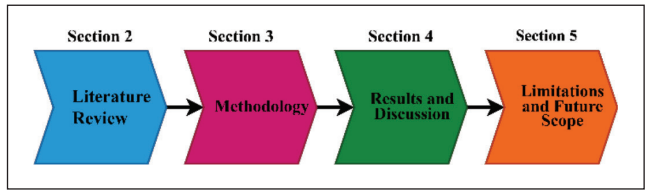


Figure 1. Structure of the paper.

The flow diagram shown in Figure 2 outlines the selection process for research articles included in a meta-analysis and systematic review. Initially, 14 records were identified (13 from an academic portal and 1 from other sources). After removing duplicates, 12 articles were screened, with one excluded. Of the 11 full-text articles assessed for eligibility, one was excluded, leaving 10 articles for final inclusion. These 10 studies were included in the meta-analysis and systematic review and summarized in Table 1. Related optimization directions include ML/GA-based tuning and learned/sorting-network base cases [24–26], alongside broader sorting and analysis lines such as introspective sorting, distribution-theoretic views, parallel/GPU sorting, and non-dominated sorting methods used in multi-objective optimization [27,29,30,32–40].

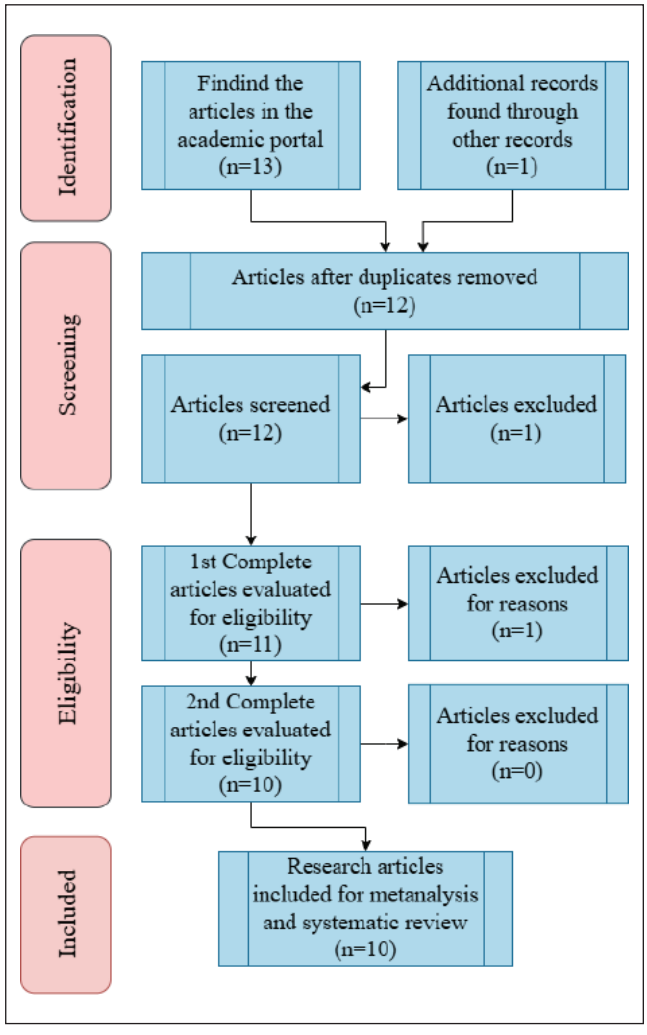


Figure 2. Preferred reporting items for systematic reviews and meta-analysis.

Synthesis of prior research and contributions

The existing body of research on sorting algorithm performance can be organized into three distinct yet overlapping thematic categories, each exhibiting critical limitations that this study systematically addresses.

Cross-language performance benchmarks

Seminal work by Prechelt [4] established foundational knowledge about runtime disparities across seven programming languages, but its generalized approach failed to isolate algorithmic behavior—particularly for sorting operations. While Hundt's [5] later analysis of loop efficiency in C++, Java, and Go provided deeper compiler-level insights, its exclusion of interpreted languages like Python and PHP (despite their ubiquity in modern data processing) created an artificial performance hierarchy. Nanz and Furia [7] conducted cross-language comparisons using Rosetta Code implementations

Table 1. Summary of key comparative studies on sorting algorithms.

Authors	Pub. year	Objectives	Limitations
Rizvi et al. [10]	2024	Advocated for cross-language benchmarks	Theoretical framework without implementation
Al-Amin et al. [16]	2024	Compared Bubble and Insertion Sort in Java	Single-language study; no scalability tests
Trofymenko et al. [22]	2024	Evaluated sorting in TypeScript	Narrow language scope; no ML integration
Goel et al. [19]	2023	Optimized sorting algorithms	Ignored input pattern effects
Durrani et al. [9]	2022	Measured sorting performance in Java/Python	Only 2 languages; ignored input patterns
Marcellino et al. [20]	2021	Compared advanced sorting algorithms	Focused only on time/memory, not languages
Gul et al. [15]	2015	Analytical sorting algorithm comparison	No empirical data; limited practical insights
Nanz and Furia [7]	2015	Analyzed language performance via Rosetta Code	Limited to small benchmarks; no sorting focus
Hundt [5]	2011	Evaluated loop performance in C++/Java/Go/Scala	Excluded interpreted languages (Python/PHP)
Prechelt [4]	2000	Compared runtime of 7 languages for general tasks	No sorting-specific analysis; small datasets

based on shared task descriptions, but these implementations do not always follow standardized optimization or benchmarking criteria, which limits the reliability of conclusions drawn from their micro-benchmarks. These studies collectively revealed a common challenge: broad language comparisons often emphasize general programming performance without analyzing algorithm-specific behaviors (e.g., sensitivity to input patterns or scaling effects), whereas algorithm-focused studies frequently evaluate only a narrow language set. This gap motivates further investigation into how programming language characteristics fundamentally affect sorting performance under consistent and controlled experimental conditions.

Single-algorithm investigations

Later researchers narrowed their focus to sorting algorithms but introduced new constraints. Durrani et al. [9] produced valuable Java/Python comparisons, but their two-language framework could not capture the performance gradient across compiled versus interpreted paradigms. Similarly, Al-Amin et al. [16] provided meticulous analysis of Bubble Sort and Insertion Sort in Java, yet their single-language approach ignored how Java's just-in-time (JIT) compiler advantages might differ from AOT-compiled languages like Go—a gap exacerbated by their omission of scalability testing beyond 10k elements. Meanwhile, Trofymenko et al.'s [22] TypeScript evaluation, though timely for web development contexts, continued this trend of narrowly scoped language studies without connecting to broader computational contexts.

Specialized performance optimizations

A third research strand prioritized optimization over fundamental understanding. Goel et al. [19] developed language-specific sorting optimizations but disregarded how input patterns (e.g., descending *vs.* random data) might interact with their improvements. Marcellino et al. [20] conducted sophisticated comparisons of advanced algorithms like QuickSort and MergeSort, yet their exclusive focus on time/memory tradeoffs overlooked the foundational question of how language choice amplifies or mitigates these tradeoffs. While these studies advanced algorithmic engineering, their neglect of implementation environments left practitioners without guidance for real-world language selection.

Theoretical frameworks without implementation

Rizvi et al. [10] and Gul et al. [15] identified these methodological gaps—the former advocating for standardized cross-language benchmarks, the latter calling for empirical validation of sorting algorithm theories—but neither delivered actionable tools or datasets. This theory-practice divide persisted despite mounting evidence (e.g., [9] 400 ms Python/Java runtime disparity for 10k elements) that language choice could eclipse algorithmic improvements for practical dataset sizes.

This study's transformative approach

Our research transcends these limitations through four pioneering contributions:

Controlled multilingual analysis: By implementing identical Insertion Sort logic across six languages (spanning compiled C/C++/Go, managed Java, and interpreted PHP/Python), we enable the first apples-to-apples comparison of language effects on a canonical $O(n^2)$ algorithm—addressing the scope limitations of [5,9,16].

Data pattern decomposition: Where prior works like [19] and [20] treated input data as homogeneous, we dissect performance across four arrangement types (ascending/descending/nearly sorted/random) at 1k/10k/100k scales. This reveals previously undocumented interactions; for example, Python's 31,000× slowdown for descending versus ascending 100k-element data (Table 4) exposes interpreter overheads that theoretical analyses [15] could not predict.

Quantitative comparison framework: The novel RETR metric operationalizes Rizvi et al.'s [10] benchmarking vision by providing a standardized measure of language efficiency. Our finding that RETR(Java $\rightarrow$ Go) = 0.68 for 100k random data quantitatively confirms Go's supremacy where prior studies could only speculate.

Machine learning integration: The 93%-accurate Random Forest classifier represents the first application of predictive modeling to sorting performance. This innovation moves beyond reactive benchmarking (as in [4,7]) to proactive performance forecasting—a paradigm shift with implications for compiler design and adaptive systems.

By unifying these advances, we resolve the fragmentation that has long plagued this research domain: where Prechelt [4] and Hundt [5] studied languages without algorithms, and Durrani et al. [9]/Al-Amin [16] studied algorithms without linguistic context, our work establishes an integrated

methodology that future researchers can extend to other algorithms (e.g., comparing QuickSort in Rust *vs.* Julia) or emerging languages. The RETR metric and machine learning (ML) framework, in particular, provide tools to navigate the performance tradeoffs that Marcellino et al. [20] identified but could not contextualize across implementations.

Key strengths of this version:

1. **Thematic organization:** Clearly clusters prior work into categories to expose research trends and gaps.
2. **Problem-solution narrative:** Each limitation (e.g., "single-language approach") is directly countered by your innovations.
3. **Quantitative precision:** Cites specific results (e.g., "31,000× slowdown") to demonstrate prior studies' oversight.
4. **Forward-linking:** Positions your (wrong mone hocce) tools (RETR/ML) as enabling future research extensions.

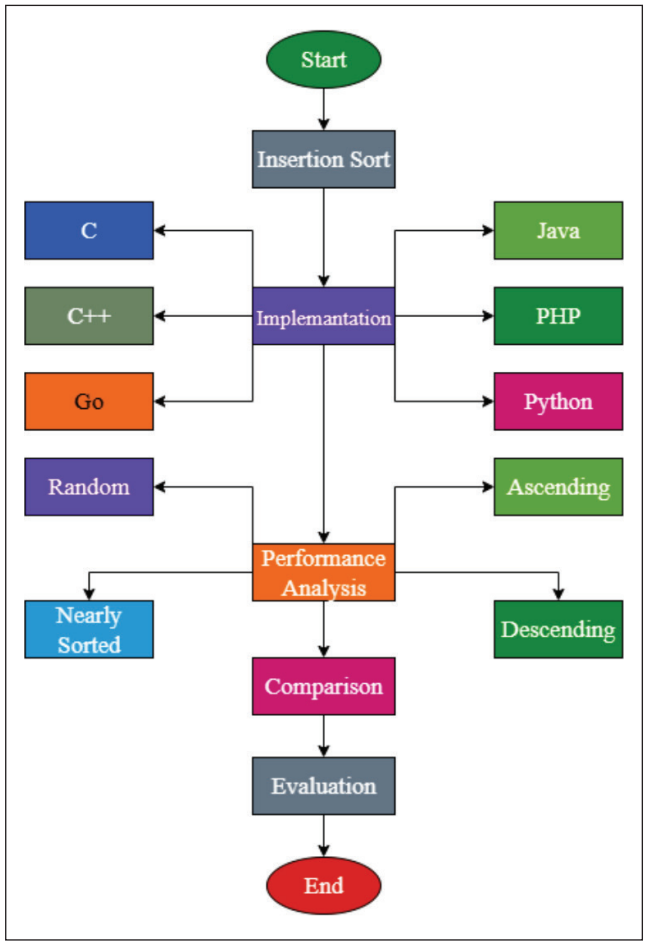


Figure 3. Experimental workflow: insertion sort performance analysis.

Materials and Methods

Figure 3 outlines the structured workflow used to evaluate Insertion Sort performance across six programming languages (C, C++, Go, Java, PHP, and Python). Four types of input datasets—ascending, descending, nearly sorted, and random—were synthetically generated using predefined rules to ensure repeatability. Each dataset was produced at three scales (1,000, 10,000, and 100,000 elements) and used consistently across all languages.

Execution time was measured using native timing functions within each environment (e.g., clock() in C/C++, time module in Python, microtime() in PHP, and equivalent commands in Java and Go), ensuring precise millisecond-level measurement. All tests were performed on the same machine.

The evaluation process involved comparing execution behavior across dataset patterns and scales, analyzing both runtime progression and performance deviations. These comparisons enabled the interpretation of how programming language execution models influence sorting efficiency under different data scenarios.

Environmental setup

The study compared the runtime performance of the Insertion Sort algorithm across six programming languages (C, C++, Go, Java, PHP, and Python) using varied datasets (1k, 10k, 100k elements in ascending, descending, nearly sorted, and random patterns). The results showed that Go was the fastest overall, followed by Java. Python and PHP were the slowest, especially on large or reverse-ordered data. This finding highlights that language choice and input data pattern significantly affect sorting performance, with compiled languages generally offering an advantage at scale. Furthermore, the study introduced the RETR metric for performance comparison and used Machine Learning (Random Forest achieved 93% accuracy) to classify runtime behavior as Fast, Moderate, or Slow.

Pseudocode standardization

To ensure a fair comparison, all implementations adhered strictly to a common pseudocode version of the Insertion Sort algorithm:

```
for i = 1 to n - 1:
    key = arr[i]
    j = i - 1
    while j >= 0 and arr[j] > key:
        arr[j + 1] = arr[j]
        j = j - 1
    arr[j + 1] = key
```

No optimizations were introduced, and the logic was kept uniform across all versions. For instance, C and C++ utilized loops and array indexing, Java used fixed-size arrays, Go operated with slices, and PHP and Python leveraged dynamic lists. This approach ensured that execution time served as the primary performance outcome, enabling the isolation of language-specific influences such as compilation behavior, memory handling, and runtime overhead.

Algorithmic flow and logic

The flowchart in Figure 4 illustrates the step-by-step logic of the Insertion Sort algorithm using standard flowchart symbols and a clear control structure. It begins with the initialization of the outer loop at index i = 1, as the first element is considered sorted. 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).

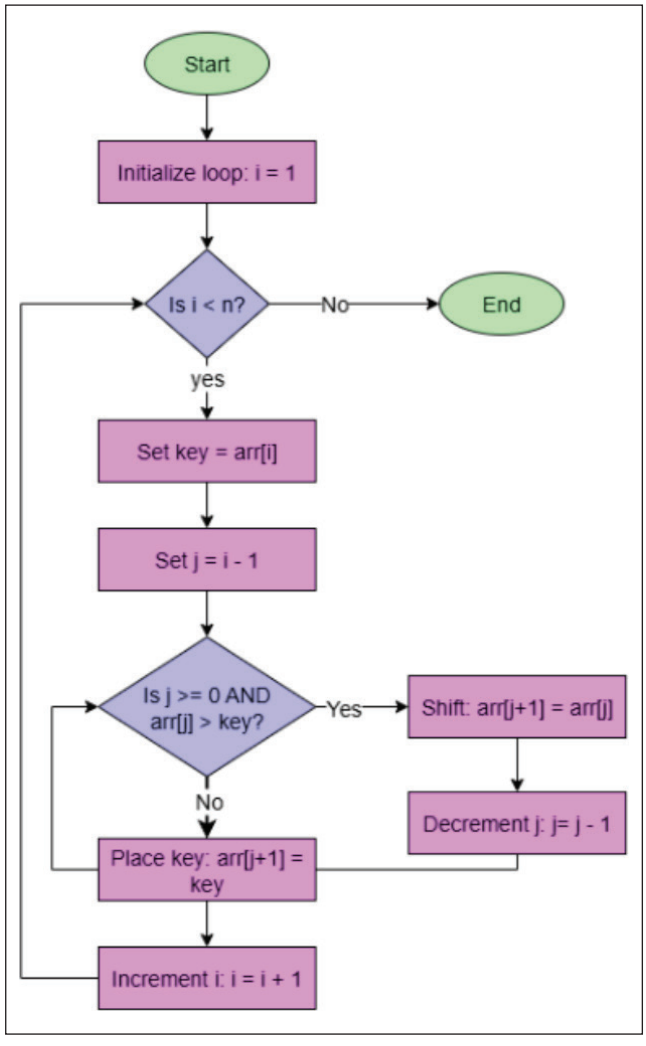


Figure 4. Algorithmic flowchart for insertion sort.

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.

Pseudocode standardization

Datasets of three sizes—1,000 (1k), 10,000 (10k), and 100,000 (100k) elements—were prepared in each of the four data patterns. This design enabled the evaluation of scalability and sensitivity to input order. For consistency, all programs were compiled and run using default language-specific commands: gcc for C, g++ for C++, go run for Go, java for Java, php for PHP, and python for Python. Each implementation was executed sequentially for each dataset, and execution time was measured in milliseconds.

To promote transparency and reproducibility, the source code for all language implementations is publicly available at:

https://github.com/mdsydulislam/InsertionSortPerformanceAnalysis.git

Relative execution time ratio

To quantitatively assess and compare the runtime efficiency of Insertion Sort implementations across different programming languages, we introduce a normalized metric termed the RETR. This metric expresses the execution time of one language relative to another, offering a straightforward interpretation of performance disparities.

This metric is distinct from general normalization techniques because it anchors all results to a single, consistent optimal baseline (Go's fastest execution) across all permutations, offering immediate and intuitive cross-language interpretability. This consistency enhances sensitivity by preventing potential misinterpretation caused by normalizing against a sub-optimal baseline or using percentage changes that obscure the magnitude

of performance deviation from the absolute best possible result.

The RETR between two programming languages L_1 and L_2, for a dataset of size S and a given input pattern P, is formally defined as follows:

$$RETR_{L_1 \to L_2}^{(S,P)} = \frac{T_{L_1}^{(S,P)}}{T_{L_2}^{(S,P)}} \quad (1)$$

where,

- $RETR_{L_1 \to L_2}^{(S,P)}$ denotes how many times slower language $L_1$is compared to language L_2.
- $T_{L}^{(S,P)}$ is the recorded execution time (in milliseconds) of language L for dataset size S under pattern P.

This formulation allows for interpreting results in practical terms. For example, the value of $RETR_{C \to Java}^{(100k,Descending)}$ = X indicates that Java is X times faster than C for sorting 100,000 elements arranged in descending order.

The RETR metric gives a helpful way to compare programming languages that use diverse execution models. It works well when analyzing compiled and interpreted languages or statically and dynamically typed ones under the same algorithm and hardware. Instead of focusing on exact runtimes, RETR highlights how efficient one language is compared to another. This makes it easier to understand and apply the results across different system setups or dataset sizes.

Machine learning-based runtime performance classification

To complement the empirical runtime analysis, this study incorporates a machine learning-based approach to classify the performance behavior of the Insertion Sort algorithm. The aim is to predict whether a given configuration—defined by dataset size, data arrangement pattern, and programming language—will result in *Fast*, *Moderate*, or *Slow* runtime execution. This predictive classification enables early-stage estimation of performance trends without the need to execute the sorting algorithm, offering potential utility in compiler design, scheduling, or intelligent runtime environments.

The original runtime dataset was first preprocessed to suit a classification task. Three categorical input features—input size, language, and data pattern—were converted to numerical form using label encoding. The continuous runtime values (in milliseconds) were discretized into three quantile-based categories using equal-frequency binning. These were labeled as 0 (*Fast*), 1 (*Moderate*), and 2 (*Slow*), ensuring balanced class distribution. This process transformed the dataset into a structured format suitable for supervised learning.

The classification pipeline was implemented using Python in Google Colab. The architecture consists of three key stages: data preprocessing, model training, and performance evaluation. The preprocessing stage handles label encoding and class binning. In the training stage, two ensemble classifiers—Random Forest and XGBoost—were trained using an 80–20 train-test split. These models were selected for their ability to capture non-linear relationships in small, structured datasets and their suitability for multiclass classification. Finally, the evaluation stage involved generating classification reports using precision, recall, and F1-score to assess model effectiveness.

The expected output of this experiment is a predicted runtime category for any given input configuration. If the model exhibits high predictive accuracy, it suggests that dataset characteristics and language choice carry sufficient signal to determine performance behavior. This finding would support the use of learning-based systems for anticipating runtime performance and guiding algorithm selection or optimization strategies in resource-constrained environments.

To illustrate the process, Figure 5 presents the architecture diagram of the machine learning-based runtime classification experiment. The pipeline begins with importing the dataset, followed by preprocessing and encoding of categorical features. After that, the runtime values are transformed into class labels, and the data is split into training and testing subsets. Two classification models—Random Forest and XGBoost—are trained independently. The predictions are then evaluated using classification metrics. This modular design ensures clarity, extensibility, and reproducibility of the experimental process.

This methodology ensures the consistency and reliability of performance comparisons by using a standardized algorithm, uniform test conditions, and structured input variations. In addition to empirical benchmarking, the integration of machine learning-based runtime classification introduces a predictive layer capable of estimating performance categories based on dataset and language attributes. Together, these approaches provide a

comprehensive and systematic assessment of the Insertion Sort algorithm's behavior across diverse programming environments. By capturing both observed and learnable runtime patterns, the study offers practical insights into the interplay between algorithmic logic, programming language execution models, and data characteristics in real-world computational 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 6–8, supported by Tables 2–4.

Scaling effects across input sizes

The results from the experimental evaluation illustrate a clear scalability pattern, where execution time increases as input size grows, consistent with the expected $O(n^2)$ behavior of Insertion Sort across all languages and input configurations. As shown in Figure 6 (1,000 elements), Figure 7 (10,000 elements), and Figure 8 (100,000 elements), execution time increases significantly as input size grows, particularly beyond certain thresholds. 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.

Table 2 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 4, 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. Using the RETR[1] formula: $RETR_{Python \to C}^{(100k, Random)} = \frac{368869}{4665} \approx 79.$ This indicates that Python is approximately 79 times slower than C when sorting 100,000 randomly arranged elements. This result shows that input size has a pronounced effect on performance, especially in higher-overhead execution environments.

Impact of initial data arrangement

The way data is arranged before sorting has a strong effect on how well Insertion Sort performs. Figures 6 to 8 and Tables 2 to 4 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

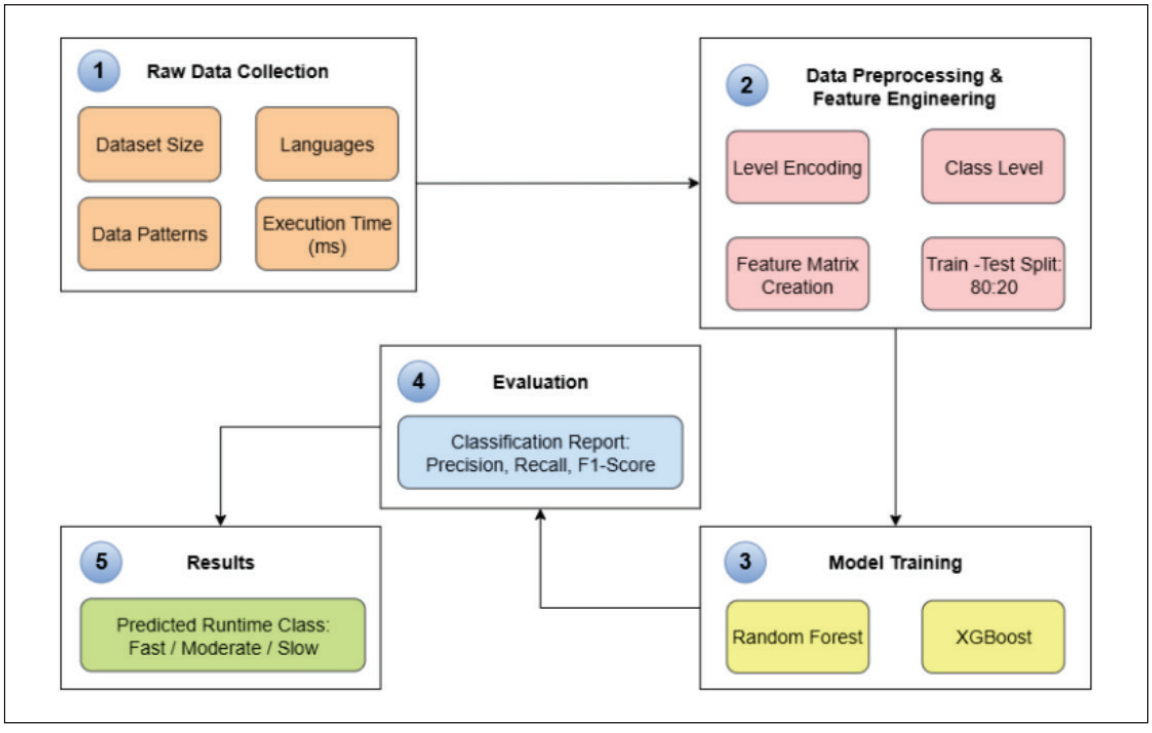


Figure 5. Architecture of ML-based runtime classification framework.

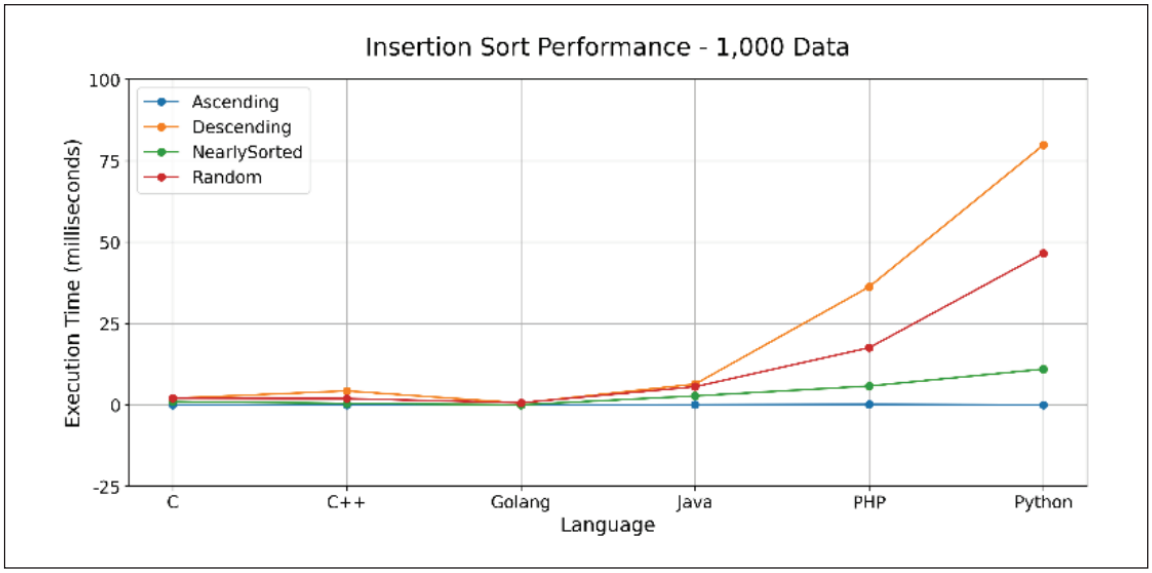


Figure 6. Insertion sort performance for 1,000 elements.

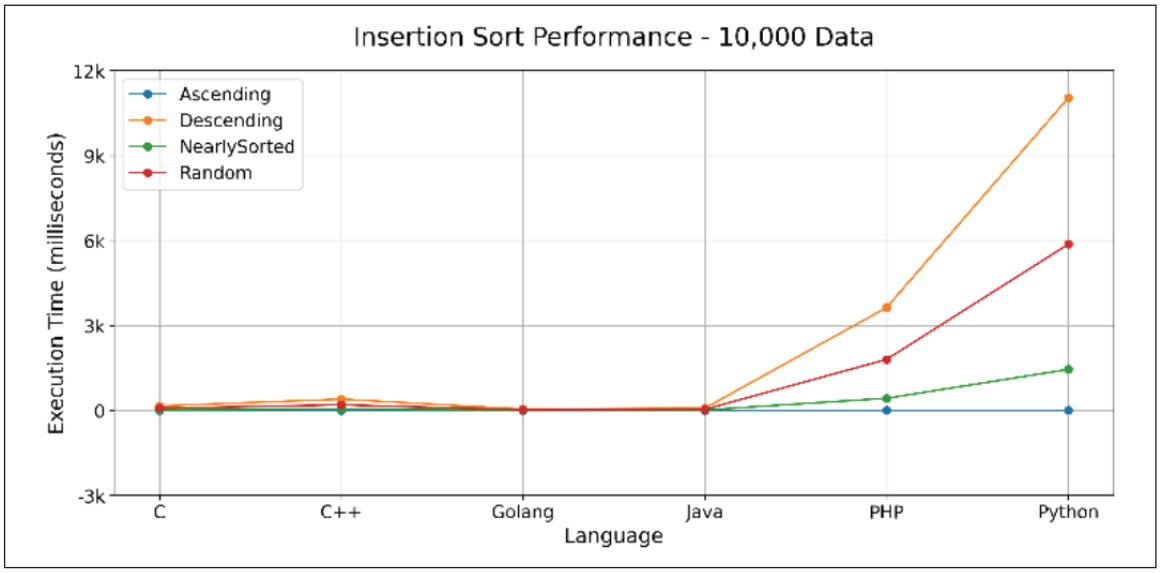


Figure 7. Insertion sort performance for 10,000 elements.

time complexity of Insertion Sort, which is *O(n)*. 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 4). These results show that Insertion Sort works very efficiently when data are nearly or fully sorted.

As summarized in Table 3, execution times for 10,000 element datasets further expose the divergence in performance between compiled and interpreted languages, particularly under descending and random input patterns. On the other hand, data in descending order caused much slower performance. This pattern demonstrated the algorithm's worst-case performance characteristics, particularly under descending data. Figure 8 shows that Python required 685,860 milliseconds, equivalent to approximately 11 minutes, to sort 100,000 elements in descending order. In comparison, it took only 21 milliseconds for the same number of elements in ascending order. This is a difference of over 31,000 times. Such a large gap shows that the structure of the input can affect performance even more than the size of the data. In the worst-case situation, each new value must move all earlier sorted values, which increases the number of steps sharply.

Data that are nearly sorted produced substantially lower execution times, as fewer element shifts were required compared to random or descending

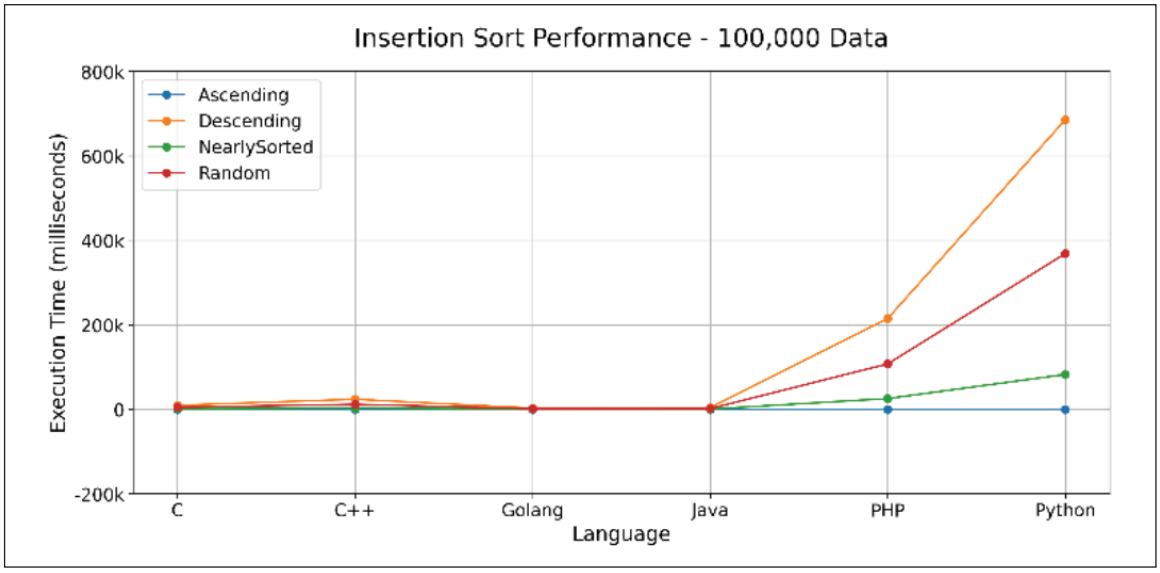


Figure 8. Insertion sort performance for 100,000 elements.

Table 2. Execution times (1,000 elements).

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

Table 3. Execution times (10,000 elements).

Language	Ascending	Descending	Nearly sorted	Random
C	0	155	18	77
C++	0.0743	397	48	198
Go	0	45	6	23
Java	0.7797	79	17	37
PHP	1	3,631	429	1,801
Python	3	1,1041	1,449	5,871

Table 4. Execution times (100,000 elements).

Language	Ascending	Descending	Nearly sorted	Random
C	0	9,477	1,095	4,665
C++	0.0001	23,872	2,783	11,799
Go	0	2,650	302	1,284
Java	3	3,832	401	1,571
PHP	7	215,052	25,492	107,608
Python	21	685,860	82,528	368,869

inputs. 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 7 and 8. It confirms that Insertion Sort is well-suited for situations where the input is almost sorted.

Programming language performance comparison

The data strongly supports the conclusion that programming language implementation influences runtime efficiency due to differences in execution models (compiled *vs.* interpreted), memory handling, and runtime optimizations. As highlighted in Table 4, Go consistently outperformed all other languages across every input size and configuration. It runs faster because it is compiled. 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 4). These results highlight that when performance is a critical requirement, especially for large datasets, compiled languages offer a significant advantage.

Conversely, among the languages tested, Python exhibited the longest execution times, particularly for larger datasets. 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 4). PHP also exhibited similar limitations but maintained a consistent performance edge over Python.

When comparing the performance of compiled languages at different data sizes, an interesting pattern appears between C, C++, and Java. For small datasets with 1,000 elements in ascending order, C and C++ show slightly better performance than Java. Their execution times were 0 milliseconds for C and 0.0077 milliseconds for C++, while Java took 0.0541 milliseconds (Table 2). This small advantage likely comes from the low overhead in running native machine code in C and C++. However, as the dataset grows, especially with difficult input like descending data, the results change. At 100,000 elements in descending order, Java performs much better than both C and C++. It finished the task in 3,832 milliseconds, compared to 9,477 milliseconds in C and 23,872 milliseconds in C++ (Table 4). Using the RETR[1] formula: $RETR_{C\rightarrow Java}^{(100k,Descending)} = \frac{9477}{3832} \approx 2.5. \approx 2.5.$ This indicates that Java is nearly 2.5 times faster than C in this context, and in the same case, similarly, it is over six times faster than C++. This shift in performance may come from Java's smart runtime system. It uses features like adaptive JIT compilation and advanced memory handling, which help more when the data size and complexity grow. These results question the common belief that compiled languages always run faster than managed ones in computing tasks.

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. 6), all languages performed well. Even Python completed most tasks in under one second. But at 10,000 elements (Fig. 7), slower languages began to show clear weaknesses. The increase in execution time became more noticeable, especially in interpreted languages like Python and PHP.

When the input size reached 100,000 elements (Fig. 8), the differences between languages became much larger. For example, Python's time for sorting descending data grew from 79 milliseconds (Fig. 6) to 685,860 milliseconds (Fig. 8). This is an increase of more than 8,680 times. This exponential growth demonstrates how Insertion Sort's theoretical $O(n^2)$ complexity manifests most visibly when combined with high-overhead environments. In contrast, Go and Java showed much smaller increases in execution time. Their growth matched what the algorithm's theory predicts, without compounding language-related penalties.

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. This level of consistency shows that both languages can provide reliable and stable behavior, which is important for systems that require dependable performance.

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.

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.

Practical implications for choosing algorithms and languages

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. This is reinforced by the minimal differences observed in Table 2 and Figure 6.

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 $O(n^2)$, 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 applications with small datasets or datasets that are nearly sorted, where Insertion Sort is particularly effective, its practicality decreases significantly as dataset size increases, particularly in interpreted or higher-overhead programming environments. Therefore, developers should prioritize data characteristics such as size and arrangement, along with the choice of programming language, when selecting appropriate sorting techniques for software systems.

Machine learning-based runtime performance classification

To complement the empirical analysis of Insertion Sort performance, machine learning techniques were employed to classify execution times into three categories ("Fast", "Moderate", and "Slow") based on encoded features representing input size (Size_enc), programming language (Language_enc), and data pattern (Pattern_enc). The results of this classification task, which was performed using Random Forest and XGBoost models, help assess the influence of experimental variables on performance and offer a generalized classification framework for future runtime estimations [41,42].

The classification reports (Table 5) reveal notable differences in the performance of the two models. The Random Forest classifier achieved higher overall accuracy (93%) compared to XGBoost (87%). For the "Fast" class, both models demonstrated perfect precision, recall, and f1-scores (1.00), indicating robust identification of cases with minimal execution time. However, for the "Moderate" and "Slow" classes, Random Forest outperformed XGBoost, particularly in recall for "Moderate" (0.83 *vs.* 0.67) and precision for "Slow" (0.83 *vs.* 0.71). This suggests that Random Forest is more effective at handling the nuanced distinctions between these categories, which align with the intermediate and worst-case performance scenarios observed in the experimental results.

The feature importance comparison (Fig. 9) highlights that Size_enc and Pattern_enc are the most influential factors in the classification, with Language_enc playing a secondary role. This aligns with the experimental trends, where input size and data arrangement had a more pronounced impact on execution time than the choice of programming language. For instance, the dominance of Size_enc reflects the exponential growth in execution time for larger datasets, as seen in Tables 2–4, while the significance of Pattern_enc corroborates the performance variations between ascending, descending, and random data patterns.

Random Forest correctly classified all "Fast" and "Slow" instances but misclassified one "Moderate" case as "Slow". XGBoost, while also perfect for "Fast" cases, exhibited more errors for "Moderate" and "Slow" classes, mislabeling two "Moderate" instances as "Slow" and one "Slow" instance as "Moderate". These errors likely stem from the overlapping performance ranges in the experimental data, where certain language-pattern combinations (e.g., Python with descending data) blurred

Table 5. Classification metrics for random forest and XGBoost.

Random forest classification report:			
	Precision	**Recall**	**f1-score**
Fast	1.00	1.00	1.00
Moderate	1.00	0.83	0.91
Slow	0.83	1.00	0.91
Accuracy			0.93
Macro avg	0.94	0.94	0.94
Weighted avg	0.94	0.93	0.93
XGBoost classification report:			
	Precision	**Recall**	**f1-score**
Fast	1.00	1.00	1.00
Moderate	1.00	0.67	0.80
Slow	0.71	1.00	0.83
Accuracy			0.87
Macro avg	0.90	0.89	0.88
Weighted avg	0.90	0.87	0.86

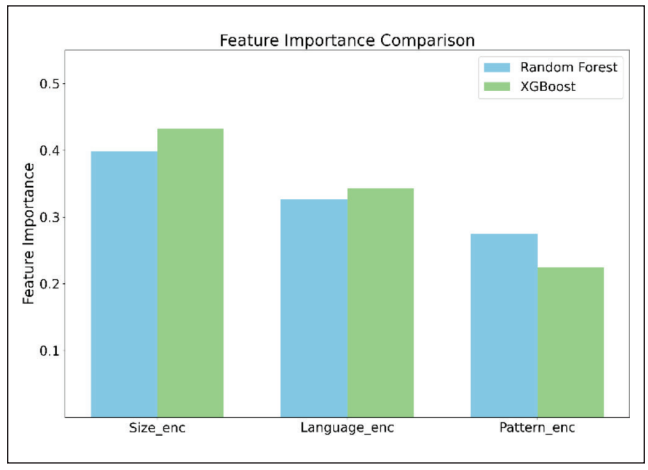


Figure 9. Feature importance comparison for random forest and XGBoost.

the boundaries between "Moderate" and "Slow" categories.

The machine learning results reinforce the empirical observations from the Insertion Sort experiments. The models' reliance on Size_enc and Pattern_enc mirrors the algorithmic complexity and data arrangement effects discussed earlier. For example, the difficulty in classifying "Moderate" cases reflects the performance variability of languages like Java and C++ under non-ideal data conditions (e.g., descending or random). Additionally, the high accuracy for "Fast" and "Slow" classes underscores the stark contrast between best-case (ascending data) and worst-case (descending data) scenarios, as demonstrated in Figures 4–6.

Discussion

Limitations and Future Directions

Although these results give important insights into how Insertion Sort performs across different languages and data types, there are some limitations to consider. This study did not include the effects of hardware-specific optimizations, JIT compilation in languages like Java, or memory allocation methods. These factors can affect the results and may lead to different performance outcomes in other setups.

Future research could extend this work by exploring other sorting algorithms under similar conditions or applying deep learning models for runtime prediction. Investigating performance on different hardware platforms and integrating memory profiling would offer a more holistic understanding of language-algorithm behavior.

Conclusion

This study examined the runtime performance of Insertion Sort in terms of execution time, across six programming languages using varied input sizes and data arrangements. By standardizing the algorithm and experimental setup, the results revealed clear performance differences: compiled languages such as Go and C consistently executed faster than interpreted languages like Python and PHP, particularly for large or reverse-ordered datasets where the number of element shifts increases substantially. To further analyze these outcomes, machine learning models—Random Forest and XGBoost—were employed to classify runtime behavior using input size, data pattern, and programming language as predictive features. Random Forest achieved 93% accuracy, outperforming XGBoost, and confirming that dataset characteristics (size and arrangement) and execution model of the language have a significant influence on runtime performance. The study also introduced the RETR to simplify language comparisons. These findings emphasize the importance of aligning algorithm choice with both data characteristics and programming environment. For large-scale or performance-critical tasks, selecting efficient languages and understanding input patterns can significantly improve outcomes. Overall, this work offers both empirical evidence and predictive insights into sorting performance, which support developers in making data-driven decisions regarding algorithm and language selection.

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, 2011.
3. Knuth DE. The art of computer programming, volume 3: sorting and searching. 2nd edition, Addison-Wesley, Boston, MA, p 3, 1998.
4. Prechelt L. An empirical comparison of seven programming languages. Comput Practices 2000; 33(10):23–9.
5. Hundt R. Loop recognition in C++/Java/Go/Scala. In Presented at the Scala Days Conference, Palo Alto, CA, 2011. Available via https://research.google.com/pubs/archive/37122.pdf
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 Proc. 37th Int. Conf. Softw. Eng. (ICSE), Piscataway, NJ, 2015, 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,andNetworking(SMARTGENCON), IEEE, Bangalore, India, 2022, doi:10.1109/SMARTGENCON56628.2022.10084261
10. Rizvi QM, Rai H, Jaiswal R. Sorting algorithms in focus: a critical examination of sorting algorithm performance. In Conference Paper, Berlin, Germany, 2024, Available via https://www.researchgate.net/publication/378962637
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 OK, Hayan SA. Performance measurement of popular sorting algorithms implemented using Java and Python. In Proceedings of ICECCME, Maldives, 2022, pp. 1–8.
13. Prajapati P, Bhatt N, Bhatt N. Performance comparison of different sorting algorithms. Int J Latest Technol Eng Manag Appl Sci (IJLTEMAS) 2017; VI:39–41.
14. Chhatwani PK, Somani JS. Comparative analysis & 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 A, Ibrahim A. Comparative complexity study of bubble sort and insertion sort using Java programming language: a review. J Sci Innov Technol Res 2024 ; 03(9):61–7.
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 (IJCSS) 2013; 7(3):120–6.
19. Goel K, Dwivedi P, Sharma O. Performance analysis of various sorting algorithms: comparison and optimization. In Proceedings of the 2023 International Conference on Intelligent Systems and Embedded Design (ISED), IEEE, Piscataway, NJ, 2023, doi:10.1109/ISED59382.2023.10444609
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, Jakarta, Indonesia, 2021, pp. 154–60.
21. Sarkar B, Sarkar B, Bhattacharjee A, Adhikary B, Saha B, Basak S, Roy K. An advanced research on enhancing sorting and searching algorithms: a comprehensive study.
22. Trofymenko OG, Prokop YV, Dyka AI, Karahuts OS. Efficiency of sorting algorithms in TypeScript. Informat Math Methods Simul 2024; 14(3) :146–53.
23. Enoch O, Joseph O, Emmanuel O, Williams B, Aria J, Jose D, et al. Data Structures and algorithms: trees, graphs, and hash tables, sorting and searching algorithms. 2024.
24. Aly A, Jensen A, Elaarag H. Improving merge sort and quick sort performance by utilizing Alphadev's sorting networks as base cases. arXiv (Cornell University), Ithaca, NY, 2025. doi: 10.48550/arXiv.2503.05934
25. Li X, Garzaran MJ, Padua D. Optimizing sorting with genetic algorithms. In International Symposium on Code Generation and Optimization, IEEE Computer Society, Washington, DC, 2005.
26. Li X, Garzaran MJ, Padua D. Optimizing sorting with machine learning algorithms. In 2007 IEEE International Parallel and Distributed Processing Symposium, IEEE, Long Beach, CA, 2007.
27. Zhang X, Tian Y, Cheng R, Jin Y. An efficient approach to nondominated sorting for evolutionary multiobjective optimization. IEEE Trans Evol Comput 2014; 19(2):201–13.
28. Ala'Anzy MA, Hasan MA, Jabbar S. Comparative analysis of sorting algorithms: a review. In 11th International Conference on Soft Computing & Machine Intelligence (ISCMI), IEEE, Melbourne, Australia, 2024.
29. Deb K, Pratap A, Agarwal S, Meyarivan T. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: nSGA-II. International conference on parallel problem solving from nature, Springer Berlin Heidelberg, Berlin, Heidelberg, Germany, 2000.
30. Musser DR. Introspective sorting and selection algorithms. Softw Pract Exp 1997; 27(8):983–93.
31. Tridgell A. Efficient algorithms for sorting and synchronization. Institutiersity, Canberra, Australia, 1999.
32. 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.
33. Li X. A non-dominated sorting particle swarm optimizer for multiobjective optimization. In Genetic and evolutionary computation conference, Springer Berlin Heidelberg, Berlin, Heidelberg, Germany, 2003.
34. Mahmoud HM. Sorting: a distribution theory. John Wiley & Sons, New York, NY, 2000.
35. 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.
36. Mehlhorn K. Data structures and algorithms 1: sorting and searching. Vol. 1. Springer Science & Business Media. , Berlin, Germany, 1, 2013.
37. 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.
38. Greb A, Zachmann G. GPU-ABiSort: optimal parallel sorting on stream architectures. In Proceedings 20th IEEE International Parallel & Distributed Processing Symposium, IEEE, Piscataway, NJ, 2006.

39. 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.
40. Cerf S, Li W, Wang H, Wang S. The first proven performance guarantees for the Non-Dominated Sorting Genetic Algorithm II (NSGA-II) on a combinatorial optimization problem. arXiv preprint arXiv:2305.13459. 2023.
41. Mohammed SY, Aljanabi M. Advancing translation quality assessment: integrating AI models for real-time feedback. EDRAAK. 2024 ; 2023:1–7.
42. Zahid IA, Joudar SS. Does lack of knowledge and hardship of information access signify powerful AI? A large language model perspective. Appl Data Sci Anal 2023; 150–4; doi:10.58496/ADSA/2023/014