

Comparison of C++ and Python performance based on selected algorithms

Kacper Burda*, Szymon Bogucki

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract

This article presents a comparative performance analysis of selected algorithms implemented in C++ and Python. The experiments were conducted in a controlled virtual environment (Ubuntu Linux running on Oracle VirtualBox with a single CPU core and 6 GB of RAM). The study covered selected sorting algorithms (Merge Sort, Quick Sort, Heap Sort), searching algorithms (Linear Search and Binary Search), and statistical measures. Tests were performed on datasets ranging from 10 to 1,000,000 elements. Performance evaluation included execution time, CPU utilization, and memory consumption. The results demonstrate a significant performance advantage of C++ in execution time and memory efficiency, particularly for large datasets, while both languages exhibit consistent scalability trends corresponding to their theoretical computational complexity.

Keywords: performance; C++; Python; algorithms; execution time

*Corresponding author

Email address: s95373@pollub.edu.pl (K. Burda)

Published under Creative Common License (CC BY 4.0 Int.)

1. Introduction

The dynamic development of information technologies has made performance-related issues one of the key areas of software engineering. The choice of a programming language can significantly affect program execution time, system resource consumption, and the scalability of a solution.

The C++ language belongs to the group of compiled languages, in which source code is translated into native machine code prior to execution. Python, in contrast, is an interpreted language that utilizes dynamic typing and high-level mechanisms that simplify implementation. These differences may lead to significant performance discrepancies, particularly when processing large datasets.

The aim of this paper is to conduct a comparative analysis of the performance of C++ and Python based on selected computational algorithms, while maintaining identical testing conditions and a consistent measurement methodology.

2. Research Objective

The objective of this study is to empirically determine the performance differences between C++ and Python based on the execution time of selected computational algorithms. The analysis includes an evaluation of the impact of dataset size on processing time, as well as a comparison of system resource utilization efficiency.

3. Literature Review

The foundations of computational complexity analysis and classical algorithms for sorting and searching have been extensively described in the algorithmic literature, including the works of Cormen et al. [1] and Sedgewick [2]. These studies provide a reference point for the theoretical evaluation of the algorithms used in this research.

In the context of the C++ language, an important role is played by the Standard Template Library (STL), which provides optimized implementations of general-

purpose algorithms [3-5]. Python, as an interpreted and dynamically typed language, offers a higher level of abstraction at the cost of additional execution overhead [6-8].

Comparative studies of programming languages indicate significant performance differences between compiled and interpreted languages, particularly in computational scenarios involving intensive data processing [9-13].

At the same time, literature on performance measurement emphasizes the importance of controlling the testing environment and conducting repeated experiments to ensure the reliability of results [14-16].

The present study contributes to this body of research by focusing on a controlled experimental environment and a broad set of algorithms representing different classes of computational complexity.

4. Research Method

The experiments were conducted in a controlled virtual environment, the configuration of which is presented in Table 1. The use of a virtual environment was intended to ensure the repeatability of experiments and to eliminate the influence of variable hardware factors, such as dynamic CPU core management or parallel execution.

Table 1: Specification of the testing environment

Operating system	Windows 10
Host RAM	8 GB
Processor	Intel Core i5-9300HF
Virtual system	Ubuntu 25.10
Virtualization environment	Oracle VirtualBox 7.2.4
Allocated RAM	6 GB
CPU cores	1 core
Architecture	x86_64

The use of a virtual machine ensured consistent experimental conditions for both implementations. In both cases, the same operating system, identical resource configuration, and the elimination of additional background processes were applied.

This approach minimizes the risk of measurement interference and increases the reliability of the obtained results.

4.1. Input Data

The test data used in the study were generated in the form of CSV files containing integer values. Each file represented a separate dataset of a specified size. The experiment included datasets of the following sizes: 10, 100, 1,000, 10,000, 100,000, and 1,000,000 elements.

The numerical values were generated randomly within the range from 1 to 10,000, which allowed for obtaining datasets with varied data distributions while maintaining the repeatability of the experimental procedure.

To ensure consistency and reliability of the results, each dataset was processed multiple times, and the obtained measurements were averaged to reduce the influence of system-related fluctuations. All input files were reused across both implementations, guaranteeing identical experimental conditions. This approach allowed for a fair comparison of performance between the programming languages while minimizing the impact of external factors such as background processes and memory management mechanisms.

4.2. Measurement Procedure

For each dataset, a set of selected algorithms was executed. To increase the reliability of the results, each experiment was performed five times, and the average value of the measurements obtained was used for further analysis.

The execution time of the algorithms was measured using high-resolution timing mechanisms available in both programming languages. In the C++ implementation, the `std::chrono::high_resolution_clock` was used to record timestamps before and after the execution of the algorithm, and the elapsed time was calculated as the difference between these two values. In the Python implementation, the `time.perf_counter()` function was used in an analogous manner, measuring the time immediately before and after the execution of the tested procedure. The difference between the recorded values was treated as the execution time of the algorithm.

Additionally, CPU usage and memory consumption were recorded automatically during program execution using system-level functions available in the operating system. In the C++ implementation, the `getrusage` function was used to obtain information about processor time (user and system time) and memory usage (`ru_maxrss`). These values were recorded immediately before and after the execution of each algorithm. CPU usage was calculated as the ratio of total processor time to wall-clock time, while memory consumption was

determined based on the maximum resident set size reported by the system. The measurements were performed automatically within the program for each algorithm run, and average values were subsequently calculated.

All measurements were conducted under identical environmental conditions, with the number of CPU cores limited to one and with additional background processes disabled. This approach minimized the influence of external factors on the experimental results.

4.3. Algorithm Implementation

As part of the experiment, a set of algorithms representing different classes of computational complexity was implemented. The study included the following groups of algorithms:

- Sorting algorithms: Merge Sort, Quick Sort, and Heap Sort,
- Searching algorithms: Linear Search and Binary Search,
- Statistical algorithms: arithmetic mean, geometric mean, harmonic mean, and standard deviation.

All algorithms were implemented in two programming languages: C++ and Python, while maintaining as similar logic and program structure as possible. The C++ implementations utilized mechanisms provided by the Standard Template Library (STL), whereas the Python implementations relied on standard language constructs. Input data were stored in array-based structures (vectors in C++ and lists in Python).

An example implementation of the Binary Search algorithm in C++ is presented in Listing 1, with the equivalent Python implementation shown in Listing 2.

Listing 1: Implementation of the Binary Search algorithm in C++

```
int binarySearch(const std::vector<int>& data, int value) {
    int left = 0, right = data.size() - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (data[mid] == value) return mid;
        if (data[mid] < value) left = mid + 1;
        else right = mid - 1;
    }
    return -1;
}
```

Listing 2: Implementation of the Binary Search algorithm in Python

```
def binary_search(data, value):
    left, right = 0, len(data) - 1
    while left <= right:
        mid = (left + right) // 2
        if data[mid] == value:
            return mid
        if data[mid] < value:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

5. Comparison of Analyzed Languages

C++ is a compiled programming language in which source code is transformed into native machine code

prior to execution [4, 5]. This allows for direct utilization of hardware resources and reduces execution overhead. Additionally, the language provides static typing and enables precise control over memory allocation.

Python is an interpreted and dynamically typed language. Code is executed by an interpreter, which introduces an additional abstraction layer between the program and the hardware [6-8]. Mechanisms such as dynamic typing and automatic memory management increase implementation flexibility; however, they may also result in higher execution overhead compared to compiled languages [9, 10].

Differences in the execution model and memory management constitute one of the key factors influencing the performance discrepancies observed in this study [17, 18].

6. Results

This section presents a concise summary of the results obtained during the experiments conducted in a controlled virtual environment. The analysis includes execution time, CPU utilization, and memory consumption for selected groups of algorithms. Particular attention is given to the relationship between dataset size and the magnitude of performance differences between C++ and Python.

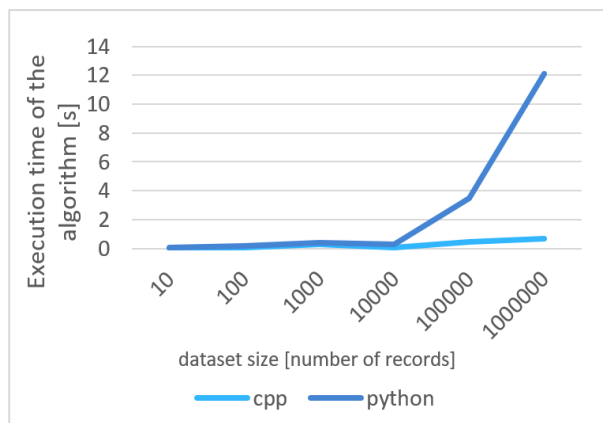


Figure 1: Execution time of the Merge Sort algorithm

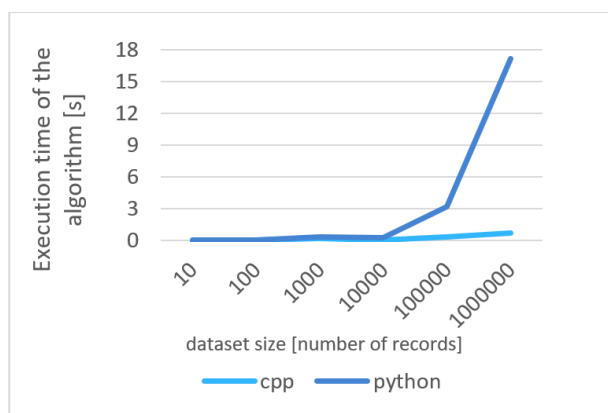


Figure 2: Execution time of the Quick Sort algorithm

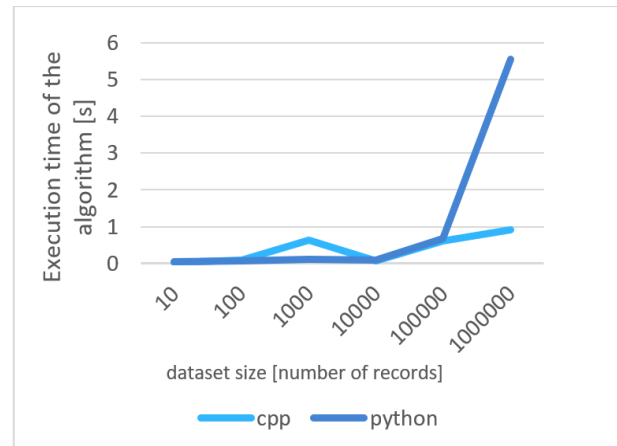


Figure 3: Execution time of the Heap Sort algorithms

The analysis of sorting algorithms demonstrated a consistent performance advantage of C++ across all dataset sizes. As the number of elements increased, the execution time differences between the implementations became increasingly pronounced, particularly for datasets containing 100,000 and 1,000,000 elements, as shown in Figures 1–3.

In both languages, the observed growth in execution time was consistent with the theoretical time complexity of $O(n \log n)$, confirming the correctness of the implementations.

Table 2: Execution time of sorting algorithms

Dataset size	Merge Sort C++ [s]	Merge Sort Python [s]	Quick Sort C++ [s]	Quick Sort Python [s]	Heap Sort C++ [s]	Heap Sort Python [s]
10	0.0147	0.0470	0.0121	0.0447	0.0180	0.0512
100	0.0736	0.1815	0.0319	0.0932	0.0862	0.0585
1000	0.2693	0.4044	0.1962	0.3423	0.6417	0.1132
10000	0.4612	0.3146	0.3451	0.2447	0.6223	0.0918
100000	0.4426	3.5518	0.3814	3.2747	0.6228	0.6626
1000000	0.6780	12.5470	0.6942	17.895	0.9669	5.5957

The execution time values for the largest analyzed dataset are presented in Table 2.

Searching algorithms exhibited significantly shorter execution times compared to sorting algorithms. Binary Search, in accordance with its logarithmic time complexity $O(\log n)$, demonstrated minimal growth in execution time as the number of elements increased. Despite the relatively small absolute values, the C++ implementation remained faster for all analyzed dataset sizes.

The analysis of searching algorithms confirms that the observed trends are consistent with theoretical computational complexity. Detailed execution times for both searching algorithms across all dataset sizes are summa-

rized in Table 3. In the case of Linear Search, execution time increases proportionally with the number of elements, corresponding to $O(n)$ complexity, as illustrated in Figure 4. In contrast, Binary Search shows only a minimal increase in execution time with growing dataset size, in line with its $O(\log n)$ characteristics.

The differences between C++ and Python implementations remain relatively constant and result from the execution model of each language. Despite the low absolute execution times, the C++ implementation consistently outperformed Python across all analyzed datasets.

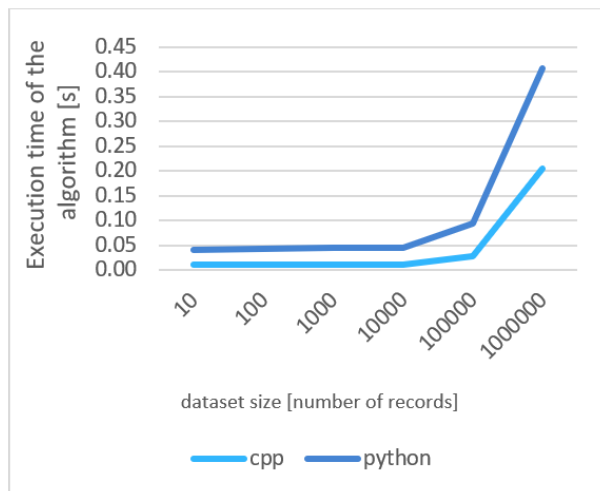


Figure 4: Execution time function for the Linear Search algorithm

Table 3: Average execution time of searching algorithms

Dataset size	Linear Search C++ [s]	Linear Search Python [s]	Binary Search C++ [s]	Binary Search Python [s]
10	0.010	0.040	0.011	0.041
100	0.011	0.043	0.011	0.041
1000	0.011	0.044	0.009	0.040
10000	0.011	0.045	0.011	0.045
100000	0.029	0.095	0.037	0.104
1000000	0.205	0.406	0.288	0.753

In the case of statistical computations, significant performance differences between the analyzed languages were observed. The execution time behavior for the standard deviation algorithm is presented in Figure 5. Algorithms computing means and standard deviation involve a large number of arithmetic operations performed within loops, which leads to the accumulation of interpretative overhead in Python.

For smaller datasets, the differences in execution time were relatively minor; however, as the dataset size increased, the performance advantage of C++ became increasingly pronounced.

The growth in execution time for the standard deviation algorithm exhibits a linear pattern, resulting from the need to process all elements in the dataset. The differences between C++ and Python implementations increase with the number of elements, indicating the accumulation of interpretative overhead in Python during intensive arithmetic operations. The observed behavior confirms that in scenarios requiring extensive floating-point computations, compiled languages achieve significantly better time efficiency.

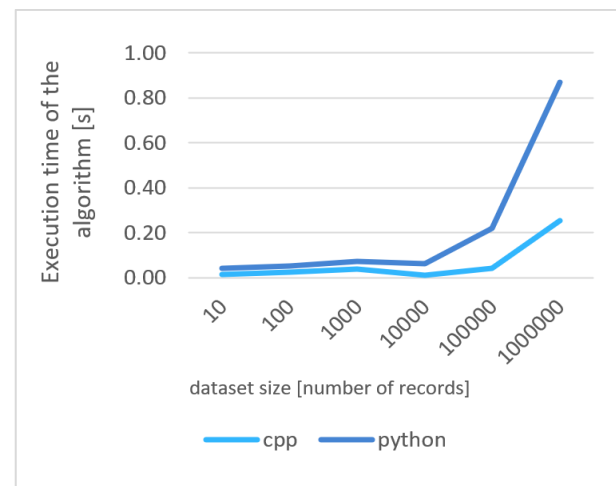


Figure 5: Execution time of the Standard Deviation algorithm

7. Conclusions

The conducted analysis confirmed significant performance differences between C++ and Python in the context of selected computational algorithms. Across all analyzed groups of algorithms, implementations in C++ consistently exhibited shorter execution times.

The most substantial differences were observed for sorting algorithms and statistical computations, particularly for large datasets. As the size of the input data increased, the execution time differences between implementations grew systematically, indicating the accumulation of interpretative overhead in Python.

The obtained results confirm that the program execution model—compilation to machine code in C++ and interpretation in Python—has a significant impact on computational efficiency, particularly in scenarios involving a large number of iterative or arithmetic operations.

At the same time, it should be noted that for small datasets, performance differences are relatively minor. In such cases, the choice of programming language may be determined by other factors, such as development speed, code readability, or ease of implementation.

The findings of this study may serve as a practical reference for developers and researchers when selecting a programming language for performance-critical applications. The results suggest that the choice between C++ and Python should be guided by the specific requirements of the application, including dataset size, computational complexity, and development constraints.

References

- [1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, MIT Press, Cambridge, 2009.
- [2] R. Sedgewick, K. Wayne, Algorithms, Addison-Wesley, Boston, 2011.
- [3] ISO/IEC, Programming Languages — C++ (ISO/IEC 14882), International Organization for Standardization, 2020.
- [4] B. Stroustrup, The C++ Programming Language, Addison-Wesley, Boston, 2013.
- [5] S. Meyers, Effective Modern C++, O'Reilly Media, Sebastopol, 2014.
- [6] Python Software Foundation, Python Documentation, <https://docs.python.org/3/>, [20.02.2026].
- [7] M. Lutz, Learning Python, O'Reilly Media, Sebastopol, 2013.
- [8] L. Ramalho, Fluent Python, O'Reilly Media, Sebastopol, 2015.
- [9] F. Dzikowski, Comparison of the performance of scripting and compiled languages: Python and C, Journal of Computer Sciences Institute 12 (2019) 1–10, <https://doi.org/10.35784/jcsi.178>.
- [10] P. Rysak, Comparative analysis of code execution time by C and Python based on selected algorithms, Journal of Computer Sciences Institute 26 (2023) 93–99, <https://doi.org/10.35784/jcsi.3109>.
- [11] P. Wlazło, J. Smółka, C++ and Java performance on the Android platform, Journal of Computer Sciences Institute 23 (2022) 135–139, <https://doi.org/10.35784/jcsi.2912>.
- [12] L. Prechelt, An empirical comparison of C, C++, Java, Perl, Python, R, and Tcl, IEEE Computer 33(6) (2000) 23–29.
- [13] S. Nanz, C. A. Furia, A comparative study of programming languages in Rosetta Code, In 37th IEEE/ACM International Conference on Software Engineering (ICSE) (2015) 778–788, <https://doi.org/10.1109/ICSE.2015.90>.
- [14] A. Georges, D. Buytaert, L. Eeckhout, Statistically rigorous Java performance evaluation, Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA) (2007) 57–76, <https://doi.org/10.1145/1297027.1297033>.
- [15] R. Jain, The Art of Computer Systems Performance Analysis, Wiley, New York, 1991.
- [16] B. Gregg, Systems Performance: Enterprise and the Cloud, Addison-Wesley, Boston, 2020.
- [17] S. Tylec, K. Woś, Swift performance in statistical applications, Journal of Computer Sciences Institute 25 (2022) 409–416, <https://doi.org/10.35784/jcsi.3067>.
- [18] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, J. Saraiva, Energy efficiency across programming languages: How do energy, time, and memory relate?, Proceedings of the 10th ACM SIGPLAN International Symposium on Software Language Engineering (SLE) (2017) 256–267, <https://doi.org/10.1145/3136014.3136031>.