JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 39 (2026) 167–175
Received: 8 April 2026
Accepted: 17 May 2026

Comparative Analysis of Chosen Programming Languages

Jakub Machnowski*, Marta Dziuba-Kozieł

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

Abstract

Programming languages continue to evolve and expand into new application domains, and it is important to determine which language is best suited for various circumstances. Among the most popular are Python, Java, and C++. We compare these three languages in terms of syntax, code length, execution speed, and resource consumption by creating three sets of programs in each language: one set performing insertion sort on lists, another set calculating numbers from the Fibonacci sequence, and a final set reading CSV files, analysing, and saving results in separate CSV files. C++ and Java demonstrate the highest performance for certain tasks, while Python remains the slowest but offers the greatest code simplicity.

Keywords: Python; Java; C++; efficiency; programming languages

*Corresponding author

Email address: jakub.machnowski@gmail.com (J. Machnowski)

1. Introduction

Programming languages continue to evolve and expand into new application domains. While some have specific applications, most are employed across many unrelated fields. Among the most popular are Python, Java, and C++ [1]. Python is known for being easy to learn and write in, but it can be slow. Java offers competitive performance due to JIT compilation and ease of cross-platform compatibility. C++ is also very fast and lightweight, but has a more complex syntax compared to the other two. All three languages have access to many external tools, numerous libraries, and dedicated communities on various forums, such as Stack Overflow.

Considering the popularity of these languages, it's important to identify the best use cases for them. The main advantages of these programming languages often influence decisions when developing an application, but recognising their downsides is also crucial for working around them. For instance, many Python packages are written in either C, C++, or Rust [2] to address the language's low speed and to avoid using native code, which tends to be very slow.

The purpose of this paper is to analyse Python, Java, and C++ languages to identify the best use cases for each. This will be accomplished by measuring the execution and build speed of identical programs written in each language, as well as their memory consumption and syntax.

2. Literature review

Performance speed analysis using lambda expressions [3] in Python, Java, and C# confirms that Python is the slowest among the languages studied. At the same time, Java ranks in the middle for execution speed. Despite its lower speed, Python currently dominates machine learning and data science [4] thanks to efficient libraries such as NumPy and Pandas, and is also used in web development and automation. Java, on the other hand, often achieves execution speeds comparable to those of C [5, 6], making it one of the most popular choices for high-performance applications, such as RESTful APIs, games, and scientific applications. It remains a popular choice for Android development despite the rise of Kotlin [7].

The C++ language is known for being lightweight and very fast, and as a result, is commonly [8] used in embedded and operational systems development, as well as in database and game programming. Both Java and Python are used for web development [9], and research shows that Java is among the fastest and most stable languages, whereas Python is among the slowest. However, the paper did not compare their syntax and code length. In terms of machine learning [10], Python performs better for processing datasets, offers better tools and community support, and its use in the field continues to grow. In contrast, Java is safer, more stable, and provides better performance. Regarding exception handling [11], Python provides simpler error messages and greater programmer freedom, while Java offers more detailed exceptions and imposes more restrictions, often with IDE warnings about potential exceptions. C++ exceptions are generally less user-friendly and often require searching for solutions in various sources, such as language documentation. Both Java and C++ can be used for mobile application development [12]. C++ generally outperforms Java in resource usage and speed, though it is considered harder to use effectively, particularly for new programmers. All studied languages were compared with respect to machine learning library development [13]. Java is the best choice for array operations, while C++ excels at calculating the dot product. Despite Python performing the worst across all measurements, it remains the most popular choice for machine learning due to its simplicity and ease of use.

3. Methods

To analyse the execution time of code written in all three investigated programming languages, a series of experiments was conducted. Three algorithms were implemented in each analysed language. These included: (I) computing the 10th, 35th, and 40th elements of the Fibonacci sequence using a recursive algorithm (Listings 4, 5, 7, 8 and 9), (II) sorting randomly generated sequences of

lengths 1,000, 50,000, and 100,000 using the insertion sort algorithm (Listings 1, 2, 3 and 6), and (III) processing CSV files by reading numerical data, performing selected computations, and saving the results to an output file (Listings 10-18).

To ensure reliable and comparable measurements of the insertion sort execution time, input sequences were generated using a fixed random seed. This approach guaranteed identical datasets across all tested implementations.

Execution times for the CSV-processing tasks were measured using three predefined input files containing 100, 10,000, and 100,000 rows, respectively. Each row consisted of five randomly generated integers in the range from 1 to 1,000. For each row, the maximum, minimum, mean, and sum values were calculated and written to a corresponding row in the output CSV file. Resource consumption was measured using Windows Performance Monitor, ensuring consistent sampling intervals and controlled background processes. Each experiment was repeated 10 times, and the average values were reported as the results. The tests were conducted on a computer equipped with an i5-1345U processor, 16 GB RAM, and Windows 11 operating system. All non-essential background processes were minimised to maintain consistent test conditions. In the presented research, the following versions of programming languages were used: Python 3.12, Java JDK 21, and C++ 20 with the following interpreters and compilers: CPython for Python and mingw-w64 13.0.0 with set -O3 optimization flag without architecture-specific optimizations. Only the internal language libraries were used during the research. The Java Virtual Machine used was the default HotSpot implementation. No additional JVM tuning parameters were applied.

3.1. Syntax

In terms of the syntax for insertion sort programs, Python is the simplest of the three languages studied (see Listings 1, 2, 4, 5, 10, 11, and 12). Java and C++ are very similar to each other, but are generally more difficult than Python.

Listing 1: Python code for performing insertion sort

```python
from random import randint
import time


def perform_insert_sort(lst):
    n = len(lst)
    for i in range(1,n):
        insert_index = i
        current_value = lst.pop(i)
        for j in range(i-1, -1, -1):
            if lst[j] > current_value:
                insert_index = j
        lst.insert(insert_index, current_value)
```

Listing 2: Python code for creating and performing insertion sort on 10 random lists

```python
for i in range(1, 11):
    lst = [randint(0, 100) for _ in range(100000)]

    start = time.time()
    perform_insert_sort(lst)
    end = time.time()
    print((end - start) * 1000)
```

Listing 3: Java code for performing insertion sort

```java
import java.lang.management.ManagementFactory;
import java.util.*;
import java.util.stream.IntStream;

public class InsertSort {
    static void perform_insert_sort(int[] arr)
    {
        for (int i = 1; i < arr.length; ++i) {
            int key = arr[i];
            int j = i - 1;
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j = j - 1;
            }
            arr[j + 1] = key;
        }
    }

    public static void main(String [] args) {
        long currentTime = System
                .currentTimeMillis();
        long vmStartTime = ManagementFactory
                .getRuntimeMXBean()
                .getStartTime();
        System.out.println("JVM start: " +
                (currentTime - vmStartTime)
        );
        for (int i = 1; i <= 10; i++) {
            int[] lst = IntStream
                    .generate(() -> new Random()
                    .nextInt(101))
                    .limit(100000)
                    .toArray();

            long start = System.nanoTime();
            perform_insert_sort(lst);
            long exec_time = System.nanoTime()-start;
            System.out.println(
                exec_time / 1_000_000.0
            );
        }
    }
}
```

For programs that generate the numbers from the Fibonacci sequence (see Listings 4, 5, 7, 8, and 9), all languages generally share similarities in syntax and readability.

Listing 4: Python function for calculating Fibonacci sequence number

```python
def perform_fibonacci(n):
    if n == 0 or n == 1:
        return 1
    return perform_fibonacci(n-1)+perform_fibonacci(
        n - 2)
```

Listing 5: Python code for collecting execution times of calculating Fibonacci sequence numbers

```python
for i in range(1, 11):
    start = time.time()
    perform_fibonacci(35)
    end = time.time()
    print((end - start) * 1000)
```

The differences in syntax are most noticeable for CSV file analysis programs (see Listings 10 – 18) where Python is the simplest and most readable, while C++ is the most complex and difficult.

Listing 6: C++ code for performing insertion sort

```cpp
#include <iostream>
#include <chrono>
#include <ctime>
#include <time.h>

using namespace std;


void insertionSort(int arr[], int n)
{
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

int main(int argc, char *argv[]) {
    using chrono::duration;
    using chrono::high_resolution_clock;
    srand(time(0));
    int lst_size = 100000;
    for (int i=1; i<=10; i++) {
        int *lst = new int[lst_size];
        for (int j=0; j<lst_size; j++) {
            lst[j] = rand() % 101;
        }

        auto start_time = high_resolution_clock::now();
        insertionSort(lst, lst_size);
        auto end_time = high_resolution_clock::now();
        duration<
            double, milli
        > exec_time = end_time - start_time;
        cout << exec_time.count() << endl;
        delete [] lst;
    }
    return 0;
}
```

Listing 7: Java code for calculating Fibonacci sequence number

```java
static int perform_fibonacci(int n) {
        if (n == 0 || n == 1) {
            return 1;
        }
        return perform_fibonacci(n-1) +
                perform_fibonacci(n-2);
    }
```

Listing 8: Java code for collecting execution times of calculating Fibonacci sequence numbers

```java
public static void main(String [] args) {
        long currentTime = System.currentTimeMillis();
        long vmStartTime = ManagementFactory
                .getRuntimeMXBean()
                .getStartTime();
        System.out.println("JVM start: " +
                (currentTime - vmStartTime));
        for (int i = 1; i <= 10; i++) {
            long start = System.nanoTime();
            perform_fibonacci(35);
            long exec_time = System.nanoTime()-start;
            System.out.println(exec_time/1_000_000.0);
        }
    }
```

Listing 9: C++ code for calculating the number from the Fibonacci sequence

```cpp
#include <iostream>
#include <chrono>
#include <ctime>
#include <time.h>

using namespace std;


static int perform_fibonacci(int n) {
    if (n == 0 || n == 1) {
        return 1;
    }
    return perform_fibonacci(n-1) +
        perform_fibonacci(n-2);
}

int main(int argc, char *argv[]) {
    using chrono::duration;
    using chrono::high_resolution_clock;
    srand(time(0));
    for (int i=1; i<=10; i++) {
        auto start_time = high_resolution_clock::now();
        perform_fibonacci(35);
        auto end_time = high_resolution_clock::now();
        duration<
            double, milli
        > exec_time = end_time - start_time;
        cout << exec_time.count() << endl;
    }
    return 0;
}
```

Listing 10: 1st part of Python function for CSV reading, analysis, and results saving

```python
def perform_csv_read_and_edit():
    with open(
        './python/result.csv', mode='w'
    ) as new_file:
        with open(
            'read_excel_test_100.csv', mode='r'
        ) as file:
            csv_file = csv.reader(file)
            max_in_file = 0
            min_in_file = 1000
            even_count = 0
            odd_count = 0
```

Listing 11: 2nd part of Python code for CSV reading, analysis, and results saving

```
for row in csv_file:
                int_row = [int(r) for r in row]
                max_in_row = max(int_row)
                min_in_row = min(int_row)
                new_file.write(f'{str(sum(
                                    int_row
                                ))},'
                               f'{str(
                                    min_in_row
                                )},'
                               f'{str(
                                    max_in_row
                                )},'
                               f'{str(mean(
                                    int_row
                                ))}\n')
                if max_in_file < max_in_row:
                    max_in_file = max_in_row
                if min_in_file > min_in_row:
                    min_in_file = min_in_row
                for number in int_row:
                    if number % 2 == 0:
                        even_count += 1
                    else:
                        odd_count += 1
            new_file.write('Largest number: '
                           f'{str(max_in_file)},'
                           ' smallest number: '
                           f'{min_in_file}\n')
            new_file.write('Even count: '
                           f'{str(even_count)},'
                           ' odd count: '
                           f'{str(odd_count)}\n')
```

Listing 12: Python code for collecting execution times of CSV reading, analysis, and results saving

```
for i in range(1, 11):
    start = time.time()
    perform_csv_read_and_edit()
    end = time.time()
    print((end - start) * 1000)
```

Listing 13: Java code for collecting execution times of CSV reading, analysis, and results saving

```
public static void main(String [] args) {
        long currentTime = System
                .currentTimeMillis();
        long vmStartTime = ManagementFactory
                .getRuntimeMXBean()
                .getStartTime();
        System.out.println("JVM start: " +
                (currentTime - vmStartTime));
        for (int i = 1; i <= 10; i++) {
            long start = System.nanoTime();
            perform_csv_read_and_edit();
            double exec_time = (double)
                (System.nanoTime()-start)/1000000;
            System.out.println(exec_time);
        }
    }
```

Listing 14: 1st part of the function in Java code for CSV reading, analysis, and results saving

```
static void perform_csv_read_and_edit()
    {
        String fileName = "./result.csv";
        File f = new File(fileName);
        try {
            f.getParentFile().mkdirs();
            f.createNewFile();
        } catch (IOException e) {
            e.printStackTrace();
        }
        int maxInFile = 0;
        int minInFile = 1000;
        int evenCount = 0;
        int oddCount = 0;
        try (BufferedWriter bw = new BufferedWriter(
                new FileWriter(fileName))) {
            try (BufferedReader br =
                new BufferedReader(new FileReader(
                    "../read_excel_test_100.csv"))) {
                String line;
                while (
                    (line = br.readLine()) != null
                ) {
                    String[] val = line.split(",");
                    List<Integer> lineData = Arrays
                            .stream(val)
                            .map(Integer::parseInt)
                            .toList();
                    int sumInRow = lineData.stream()
                            .mapToInt(v -> v).sum();
                    int maxInRow = lineData.stream()
                            .mapToInt(v -> v).max()
                            .orElseThrow(
                                Exception::new
                            );
                    int minInRow = lineData.stream()
                            .mapToInt(v -> v).min()
                            .orElseThrow(
                                Exception::new
                            );
                    double meanInRow = (double)
                        sumInRow / 5;
                    bw.write(
                        String.valueOf(sumInRow) +
                        ',' + minInRow + ',' +
                        maxInRow + ',' + meanInRow
                    );
                    bw.newLine();
                    if (maxInFile < maxInRow) {
                        maxInFile = maxInRow;
                    }
                    if (minInFile > minInRow) {
                        minInFile = minInRow;
                    }
                    for (Integer number : lineData) {
                        if (number % 2 == 0) {
                            evenCount++;
                        } else {
                            oddCount++;
                        }
                    }
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
```

Listing 15: 2nd part of function in Java code for CSV reading, analysis, and results saving

```
bw.write(
                "Largest number: " + maxInFile +
                ", smallest number: " + minInFile
            );
            bw.newLine();
            bw.write(
                "Even count: " + evenCount +
                ", odd count: " + oddCount
            );
            bw.newLine();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

Listing 16: 1st part of function in C++ for collecting execution times of CSV reading, analysis, and results saving

```
void readAndEditCsv()
{
    ifstream file("../read_excel_test_100.csv");
    ofstream output_file;
    output_file.open("result.csv");
    if (!file.is_open()) {
        cerr << "Failed to open file" << endl;
        return;
    }
    string line;
    int max_in_file = 0;
    int min_in_file = 1000;
    int even_count = 0;
    int odd_count = 0;
    while (getline(file, line)) {
        stringstream ss(line);
        string cell;
        int sum = 0;
        int max_in_row = 0;
        int min_in_row = 1000;
        while (getline(ss, cell, ',')) {
            int int_cell = stoi(cell);
            if (int_cell > max_in_row) {
                max_in_row = int_cell;
            }
            if (int_cell < min_in_row) {
                min_in_row = int_cell;
            }
            if (int_cell % 2 == 0) {
                even_count++;
            } else {
                odd_count++;
            }
            sum += int_cell;
        }
        output_file << to_string(sum) << "," <<
            to_string(min_in_row) << "," <<
            to_string(max_in_row) << "," <<
            to_string((double) sum / 5) << "\n";
        if (max_in_file < max_in_row) {
            max_in_file = max_in_row;
        }
        if (min_in_file > min_in_row) {
            min_in_file = min_in_row;
        }
    }
```

Listing 17: 2nd part of function in C++ for collecting execution times of CSV reading, analysis, and results saving

```
output_file << "Largest number: " <<
        to_string(max_in_file) <<
        ", smallest number: " <<
        to_string(min_in_file) << endl;
    output_file << "Even count: " <<
        to_string(even_count) <<
        ", odd count: " <<
        to_string(odd_count) << endl;
    file.close();
    output_file.close();
}
```

Listing 18: C++ code for collecting execution times of CSV reading, analysis, and results saving

```
int main(int argc, char *argv[]) {
    using chrono::duration;
    for (int i=1; i<=10; i++) {
        auto start_time = chrono::steady_clock::now();
        readAndEditCsv();
        auto end_time = chrono::steady_clock::now();
        duration<
            double, milli
        > exec_time = end_time - start_time;
        cout << exec_time.count() << endl;
    }
    return 0;
}
```

In terms of code length (see Table 1), Python algorithms are often the shortest of the three researched languages, while C++ programs are the longest, with Python code sometimes being nearly twice as concise as the other two. Java code is usually slightly shorter than C++ code.

The analysis of code complexity was limited to line counts, which does not fully capture software complexity. More advanced metrics such as cyclomatic complexity or cognitive complexity could provide a more comprehensive evaluation.

Table 1: Total code length comparison [lines]

Language	Python	Java	C++
Insertion sort	22	39	42
Fibonacci	14	23	28
CSV files	39	88	75

3.2. Execution time

As shown in Figures 1, 3, and 5, the time of insertion sort algorithm execution in Java and C++ is comparable in the case of lists having more than one thousand elements. However, C++ yields consistent results, whereas the times obtained in Java can vary. Code written in C++ performs better for individual code execution on small lists, e.g., 1,000-element long. Python was the slowest of the three, as depicted in Figures 2, 4, and 6; however, it exhibited higher execution times consistent with its interpreted execution model.

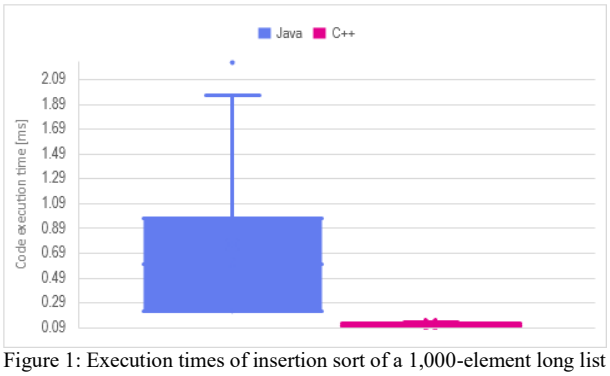

Figure 1: Execution times of insertion sort of a 1,000-element long list for Java and C++ languages

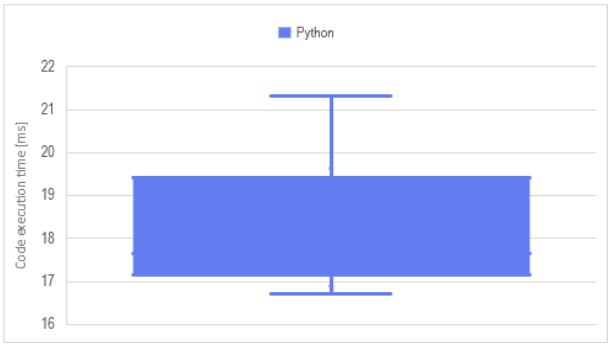

Figure 2: Execution times of insertion sort of a 1,000-element-long list for the Python language

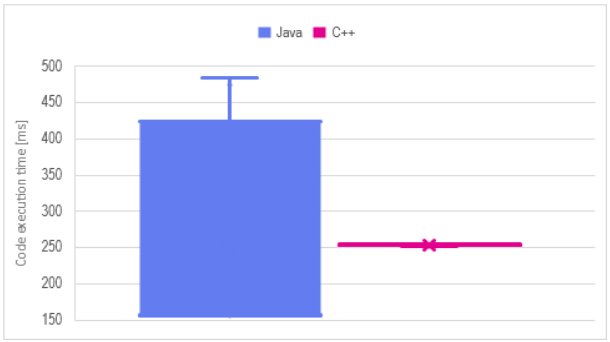

Figure 3: Execution times of insertion sort of a 50,000-element-long list for Java and C++ languages

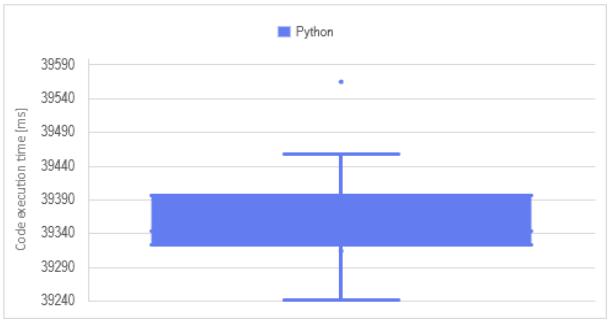

Figure 4: Execution times of insertion sort of a 50,000-element-long list for the Python language

Overall, C++ demonstrated better performance than Java in terms of build time and total execution time. It had slower individual execution times later on, while also being more consistent (see Tables 2, 3, and 4), due to the JVM, which introduces startup overhead, but JIT compilation may improve performance during longer executions.

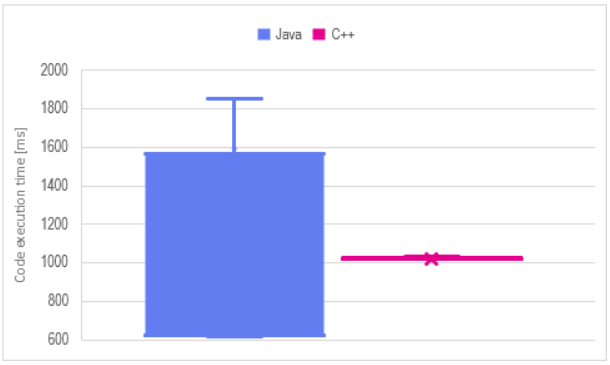

Figure 5: Execution times of insertion sort of a 100,000-element long list for Java and C++ languages

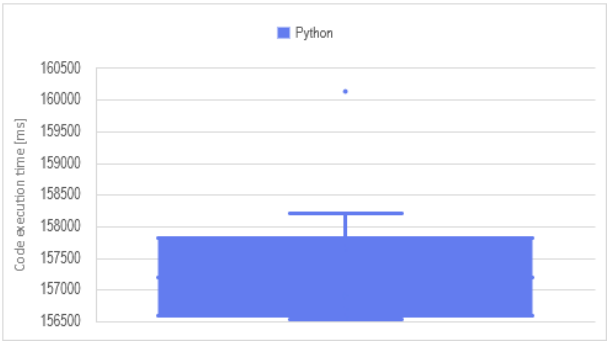

Figure 6: Execution times of insertion sort of a 100,000-element-long list for the Python language

Table 2: Comparison of execution times for performing insertion sort on 1,000-element-long lists

Language	Python	Java	C++
Average [ms]	18.17	0.76	0.11
Deviation [ms]	1.46	0.73	0.01
Build time [ms]	-	1305	0
JVM [ms]	-	58	-

Table 3: Comparison of execution times for performing insertion sort on 50,000-element-long lists

Language	Python	Java	C++
Average [ms]	39365.66	246.82	253.69
Deviation [ms]	88.51	145.32	1.36
Build time [ms]	-	1305	0
JVM [ms]	-	58	-

Table 4: Comparison of execution times for performing insertion sort on 100,000-element long lists

Language	Python	Java	C++
Average [ms]	157452.33	936.92	1023.28
Deviation [ms]	1109.18	510.70	6.84
Build time [ms]	-	1305	0
JVM [ms]	-	58	-

In the case of Fibonacci sequence calculations using recurrence (see Figures 7, 8, 9, 10, and 11), Java was the fastest for calculating the 40th number. Still, for lower numbers, C++ was faster, while Python was extremely slow. This time, build times and JVM (see Tables 5, 6, and 7) startup didn't significantly affect the total execution time.

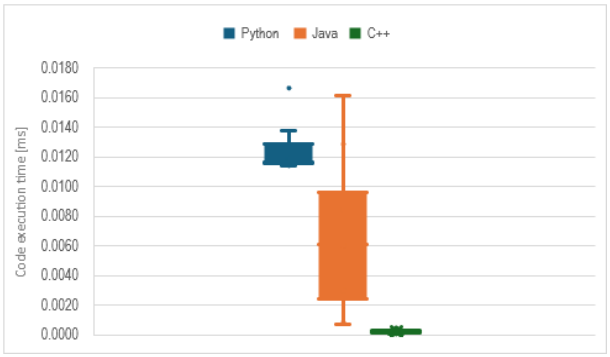

Figure 7: Execution times of Fibonacci sequence 10th number calculation for Python, C++, and Java languages

Figure 8: Execution times of the calculation of the 35th number of the Fibonacci sequence for C++ and Java languages

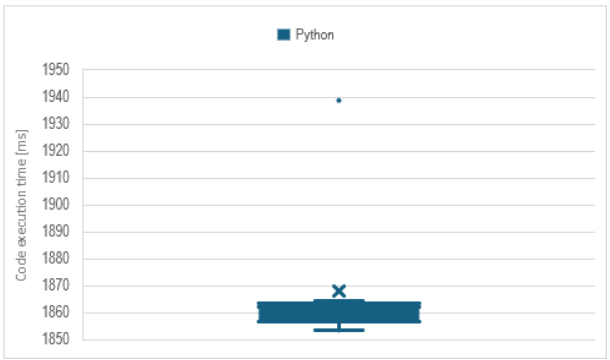

Figure 9: Execution times of the Fibonacci sequence 35th number calculation for the Python language

Table 5: Comparison of execution times for finding the 10th number from the Fibonacci sequence

Language	Python	Java	C++
Average [ms]	0.0125	0.0067	0.0002
Deviation [ms]	0.0017	0.0049	0.0001
Build time [ms]	-	3000	0
JVM [ms]	-	53	-

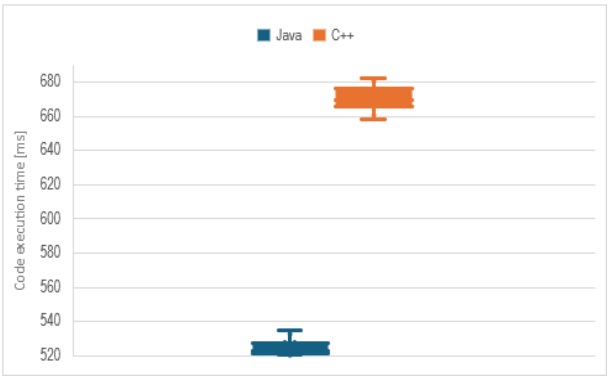

Figure 10: Execution times of the calculation of the 35th number of the Fibonacci sequence for C++ and Java languages

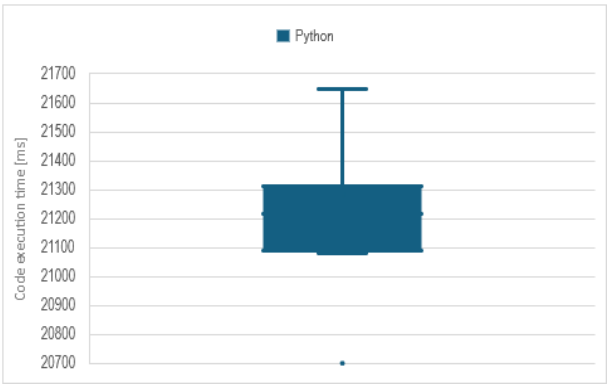

Figure 11: Execution times of the Fibonacci sequence 40th number calculation for the Python language

Table 6: Comparison of execution times for finding the 35th number from the Fibonacci sequence

Language	Python	Java	C++
Average [ms]	1867.96	48.22	12.81
Deviation [ms]	25.15	0.30	0.13
Build time [ms]	-	3000	0
JVM [ms]	-	53	-

Table 7: Comparison of execution times for finding the 40th number from the Fibonacci sequence

Language	Python	Java	C++
Average [ms]	21208.09	524.86	670.33
Deviation [ms]	237.96	5.02	7.59
Build time [ms]	-	3000	0
JVM [ms]	-	53	-

Results for programs that read and analyse CSV files are shown in Figures 12, 13, and 14. C++ was the fastest among the three languages for smaller files, with Java being the fastest for 10,000- and 100,000-row files. Python was generally the slowest language for individual executions.

Similar to previous experiments, Java performance is influenced by JVM startup overhead and its longer build time (see Tables 8, 9, and 10), resulting in Java ultimately being the slowest of the three languages studied.

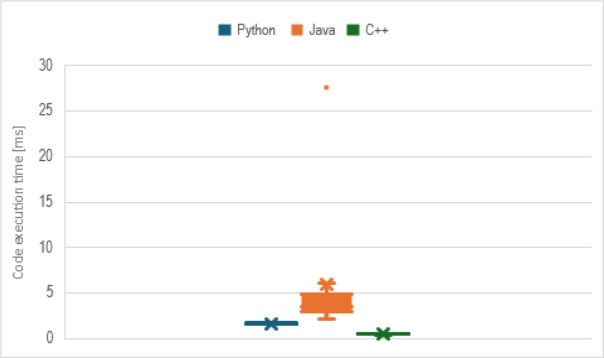

Figure 12: Execution times of reading a 100-row-long CSV file, analysing, and saving results for Python, Java, and C++ languages

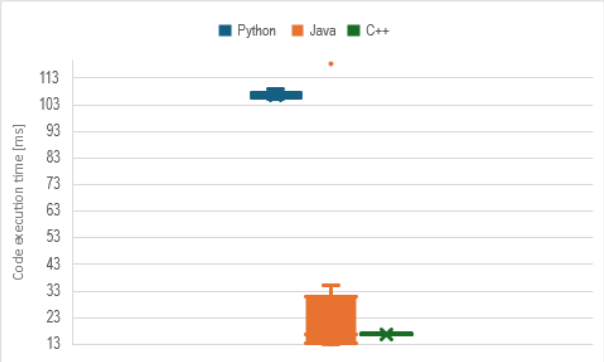

Figure 13: Execution times of reading a 10,000-row-long CSV file, analysing and saving results for Python, Java, and C++ languages

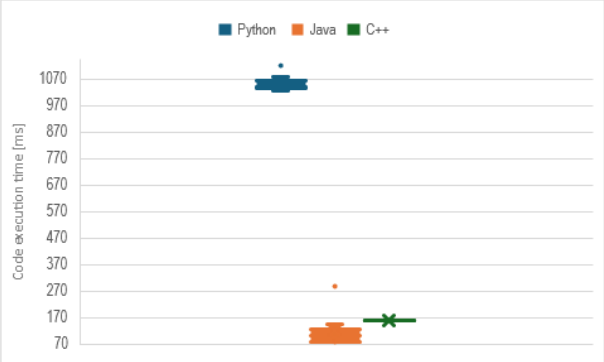

Figure 14: Execution times of reading a 100,000-row-long CSV file, analysing, and saving results for Python, Java, and C++ languages

Table 8: Execution times for performing CSV file analysis of a 100-row-long file

Language	Python	Java	C++
Average [ms]	1.67	6.01	0.52
Deviation [ms]	0.14	7.67	0.09
Build time [ms]	-	355	0
JVM [ms]	-	48	-

Table 9: Execution times for performing CSV file analysis of a 10,000-row-long file

Language	Python	Java	C++
Average [ms]	106.72	28.93	16.96
Deviation [ms]	1.19	32.36	0.14
Build time [ms]	-	355	0
JVM [ms]	-	48	-

Table 10: Execution times for performing CSV file analysis of a 100,000-row-long file

Language	Python	Java	C++
Average [ms]	1055.63	121.07	161.63
Deviation [ms]	28.58	63.65	2.01
Build time [ms]	-	355	0
JVM [ms]	-	48	-

3.3. Resource consumption

Regarding resource consumption for programs inserting a 10,000-element list (see Table 11), C++ was the most efficient. In contrast, Python performed the worst in terms of processor usage, and Java performed the worst in terms of memory consumption.

Table 11: Comparison of resource consumption for insertion sort programs for a 10,000-element long list

Language	Python	Java	C++
Memory [MB]	5	70	0.4
Processor [%]	23	15	1

In terms of resource consumption for programs that compute the 40th Fibonacci number (see Table 12), C++ performed best, while Java was the least efficient. Python's resource usage was in the middle.

Table 12: Comparison of resource consumption for programs finding the 40th number in the Fibonacci sequence

Language	Python	Java	C++
Memory [MB]	4.5	28	0.2
Processor [%]	20	20	6

4. Limitations

There are also several limitations of the study. Only three algorithms were selected for the research and there may be algorithms better suited for comparison of code length, performance and resource usage.

For Java language JVM warmup time was included in overall time performance, which in some cases altered the results in favour of C++ language. In the case that all Java algorithms were performed in the same code execution, the warmup would happen only once. There are also tools such as CRaC, to store the warmup state and reduce its impact on execution time of algorithms.

Insertion sort and Fibonacci sequence algorithms also prioritized time complexity, $O(n^2)$ and $O(2^n)$ respectively. As such, researching these algorithms prioritized analysing their inefficiencies, rather than language characteristics. It should be emphasized that the selected algorithms (recursive Fibonacci and insertion sort) are not representative of real-world software performance. The recursive Fibonacci implementation primarily measures function call overhead rather than computational efficiency, while insertion sort represents a quadratic-time algorithm rarely used in production systems for large datasets.

Finally, the statistical testing was limited to the average values and deviations of execution times. Further analysis, including calculating variance, confidence intervals, and significance testing, could present differences in results in a more detailed way. Moreover, it cannot be determined whether the observed differences between programming languages are statistically significant or fall within measurement variability.

All experiments were conducted on a single hardware configuration without parallel or multithreaded execution. Therefore, the results may not generalize to other hardware platforms or concurrent workloads.

5. Conclusions

The study compared Python, Java, and C++ in terms of execution time, resource consumption, and code complexity using three representative programming tasks: insertion sort, Fibonacci sequence calculation, and CSV file processing. The results indicate that C++ generally provides the highest performance and lowest resource consumption, particularly for tasks involving intensive computation and data processing. This confirms its suitability for systems where efficiency and low-level control are critical. However, this advantage comes at the cost of greater code complexity and reduced readability, which may increase development time and maintenance effort.

Java performs well, especially in tasks such as sorting and numerical calculations. Although the Java Virtual Machine adds some startup delay, its effect on overall performance is less noticeable in longer-running applications. Java also provides a good balance between performance and code maintainability, making it ideal for large-scale, cross-platform systems.

Python, although consistently slower and more resource-consuming in the conducted experiments, provides the highest level of code simplicity and readability. Its concise syntax allows for rapid development and easier maintenance, which explains its widespread use in fields such as data analysis, machine learning, and scripting. In practice, performance issues are often addressed by using optimised external libraries written in lower-level languages.

Performance depends on the task type and the execution model (interpreted, JIT, or compiled). No single language was consistently the fastest across all benchmarks.

It should be noted that the presented results depend on the specific implementation details, problem sizes, and measurement methods used in this study. Therefore, the findings should be viewed as indicative rather than conclusive, and further experiments with a wider range of benchmarks and more rigorous measurement techniques are recommended.

In conclusion, choosing a programming language should depend on the specific needs of the application, such as performance, development speed, ease of maintenance, and available ecosystem support. No single language is the best for all situations; each has unique advantages that make it suitable for different types of problems.

References

[1] TIOBE Index for April 2026, https://www.tiobe.com/tiobe-index/, [30.04.2026].

[2] P. D. Johnson, D. D. Hodson, PyO3: Building Python Extension Modules in Native Rust with Performance and Safety in Mind, Communications in computer and information science 2258 (2025) 23-30, https://doi.org/10.1007/978-3-031-85902-1_4.

[3] T. Todorov, N. Noev, Performance of lambda expressions in high level programming languages, TEM Journal 12(4) (2023) 2235–2240, https://doi.org/10.18421/tem124-34.

[4] S. Raschka, J. Patterson, C. Nolet, Machine Learning in Python: Main Developments and Technology Trends in Data Science, Machine Learning, and Artificial Intelligence 11(4) (2020) 193, https://doi.org/10.3390/info11040193.

[5] B. Oancea, I. G. Rosca, T. Andrei, A. I. Iacob, Evaluating Java performance for linear algebra numerical computations, Procedia Computer Science 3 (2011) 474–478, https://doi.org/10.1016/j.procs.2010.12.080.

[6] G. Nikishkov, Y. Nikishkov, V. Savchenko, Comparison of C and Java performance in finite element computations, Computers & Structures 81(24–25) (2003) 2401–2408, https://doi.org/10.1016/s0045-7949(03)00301-8.

[7] L. Ardito, R. Coppola, G. Malnati, M. Torchiano, Effectiveness of Kotlin vs. Java in android app development tasks, Information and Software Technology 127 (2020) 106374, https://doi.org/10.1016/j.infsof.2020.106374.

[8] ByteByteGo Incorporated. ByteByteGo | Top 8 C++ use cases. ByteByteGo, https://bytebytego.com/guides/top-8-c-use-cases/, [15.03.2026].

[9] E. Kemer, R. Samli, Performance comparison of scalable rest application programming interfaces in different platforms, Computer Standards & Interfaces 66 (2019) 103355, https://doi.org/10.1016/j.csi.2019.05.001.

[10] Python vs. Java in AI (Late 2024): A Comprehensive Ecosystem Comparison with Scoring and Evaluation, https://bayramblog.medium.com/python-vs-java-in-ai-late-2024-a-comprehensive-ecosystem-comparison-with-scoring-and-evaluation-999a6896c4b0, [28.09.2025].

[11] V. Shah, Java vs. Python: A Head-to-Head Coding Comparison (Part 2): Exception Handling. Medium, https://medium.com/@vishal7090/java-vs-python-a-head-to-head-coding-comparison-part-2-exception-handling-e0d013161458, [28.09.2025].

[12] P. Wlazło, J. Smołka, C++ and Java performance on the Android platform, Journal of Computer Sciences Institute 23 (2022) 135–139, https://doi.org/10.35784/jcsi.2912.

[13] O. C. R. Rachmawati, A. R. Barakbah, T. Karlita, Programming language selection for the development of deep learning library, JOIV International Journal on Informatics Visualization 8(1) (2024) 434, https://doi.org/10.62527/joiv.8.1.2437.