Measuring the correlation between type system and performance

What is the correlation between the type system of a programming language and the performance of the program?

Arne Vercauteren
6 Informatica- en communicatiewetenschappen
School year 2024-2025 | VLOT! Campus Sint-Laurentius
Supervising teacher: Wim Baert

Contents

1 Abstract

This paper details a research project spanning over a school year, it describes the entire process, along with the results and conclusions. The project aimed to determine the correlation between type system and performance. The results indicate a relatively mild correlation in favor of static languages for performance. There were a number of tests done to draw this conclusion, the main ones, however, were the correlation calculations. The correlation was calculated in 2 ways, the first was the Point-Biserial correlation. This test indicated a correlation with medium effect size, but was not particularly statistically significant according to the resulting p-value. This statistical insignificance was likely due to the effect of outliers on the distribution of raw data and the non-normal distribution of that data. To counteract this, the Rank-Biserial correlation (which is influenced far less by outliers) was calculated as well, which resulted in a very similar effect size, but with an extremely low p-value, confirming its significance. The Rank-Biserial correlation therefore allowed for a more robust conclusion, and the combination of these methods provides more insight into the results of this project. However, this project was subject to a number of limitations (mostly in skill, experience, knowledge, scale, time, and resources), which, by extension, allow for various valid criticisms. The main problem being the small sample size of only 14 languages: 7 static and 7 dynamic. The fact that only the execution time per language was measured, as opposed to a combination of metrics, can also be called into question. This paper also describes several additional analyses which help clarify the relationship between type system and performance.

2 Introduction

2.1 Motivation and Practicality

My name is Arne Verauteren and I study computer science, the subject of this paper – type systems – was chosen because I have a great personal interest in programming languages and their design and functionality.

In real-world, large-scale systems, performance is of incredible importance; choosing the right framework for your system can entail massive benefit when done right, or great loss when done wrong. In large corporations, for example, making the wrong choice can cause massive economic strain; while making the right choice can give you an edge over the competition. Performance is important, not only for developers, but also for consumers; smoother experiences, shorter waiting times during startups and quicker web pages are just a few examples. By measuring the correlation between type system and performance, this paper attempts to help computer science at large in obtaining a clearer view of what the benefits of static and dynamic type systems are and therefore in making the right choice between them considering the specific use case and requirements (although it only touches on a single aspect of what influences such a choice, namely, the performance).

2.2 Concept details

What is a type? A type is a classification or group, like a type of physical object or a type of abstract idea. You can also have types within other types. For example, a dog is a type of animal, so is a cat, an animal is a type of living being. Nine is a number, specifically a natural number, a natural number is a type of number, and a number is a type of character. In programming, types are used to define not only what exactly something is, but also how we can use it, it has what we call an interface. In practice, an interface is a set of functions; a dog can bark; a cat can meow; but they can both eat. So we would give both the cat type and the dog type separate sound functions, but the type animal would get an eat function. Type systems are used to make the language make sense; the type system will throw an exception when you try to count with dogs or make a number meow.

What is a type system? "A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute." (Pierce, 2002, p. 1) There are many distinctions one could make between type systems. For this project, the sole distinction will be between static and dynamic type systems. In dynamic type systems, type checking is performed at runtime and variables can change types throughout the program. In static type systems, type checking is performed during compile time and variables are generally not allowed to change types throughout the program. (GeeksForGeeks, 2019)

What does performance mean in this context? 'Performance' could be interpreted in a wide variety of ways; we could measure execution time, but also memory usage, compilation speed (for compiled languages), or even energy efficiency. To keep this project manageable, we will take the simple (and most obvious) approach and only measure execution time.

Considering the audience for this paper and the low-level academic nature of it, some topics and concepts that might not usually need to be explained in a professional paper will be given a short explanation here.

A big thank you to the supervising teachers for their support of this project.

3 Research

3.1 Program and Language Selection/Homogenization

To accurately measure the correlation, 14 languages were chosen, and 7 albeit simple programs were implemented in every language, resulting in a total of 98 unique source code files. The ideas for the implemented programs came from rosettacode.org. Rosetta code is an online database containing thousands of code snippets: Crowdsourced programs implemented in many languages. Although the selection of programs came from Rosetta, the actual implementations were written by AI and then verified, and, if necessary, edited by the researcher.

Below is a list of the selected programs and languages:

Languages:

Dynamic

- Python
- Javascript
- PHP
- Perl
- Lua
- Julia
- Awk

Static

- C
- C#
- Swift
- Go
- Java
- Rust
- Kotlin

Programs:

- 100 doors x10000
- factorial with n = 10 x100000
- fibonacci sequence with n = 10 x100000
- fizzbuzz x10000
- reverse a string using ”Hello, World!” x100000
- selection sort using {29, 10, 14, 37, 13, 54, 76, 98, 12, 1, 64, 13, 512} x100000
- sieve of eratosthenes using n = 10000 x10000

For-loops were wrapped around each program to obtain practical execution times (so as to avoid measuring in microseconds or nanoseconds for some languages), the numbers next to the programs correspond to the number of iterations.

The process of homogenizing such programs across different languages is one that comes with a plethora of challenges. Some languages have similar syntax, but include different operations within the (seemingly) equivalent function, some do things automatically that others don’t. Even though the programs used in this research were quite simple, and even though the chosen languages had very similar syntax, there are still countless corner cases one could encounter. Not to mention the amount of trouble that would be encountered if the languages and programs were to be selected randomly.

3.2 Program Descriptions

100 doors creates an array of 100 booleans (doors), sets them all to false, then iterates over all of them and toggles them, then does the same thing but only with every 2nd bool, then every 3rd and so on, until it only toggles the 100th bool, then it prints out every bool’s state.

Factorial calculates the factorial of n iteratively, and then prints it, 10 is used as n here.

Fibonacci sequence calculates and prints fibonacci numbers up to the n-th number recursively, 10 is used as n here.

FizzBuzz writes each number from 1 to 100, but writes Fizz when the number is divisible by 3, Buzz when it's divisible by 5, and FizzBuzz when it's divisible by 15.

Reverse a string reverses a string and then prints both the original and the reversed string; "Hello, World!" is used here.

Selection sort performs a selection sort on a given array of integers, and prints out the result, this array is used here : {29, 10, 14, 37, 13, 54, 76, 98, 12, 1, 64, 13, 512}

Sieve of eratosthenes calculates all prime numbers up to n, 10000 is used as n here.

The C# versions of each program have been provided in the appendix of this paper as examples. All of the inputs have been hardcoded for ease of automation.

3.3 Methodological Limitations

Because of the limited time, resources, and scope of this project, along with the limitations in skill, knowledge, and experience on the part of the researcher, there were certain limitations to the research method. The programs chosen for this project were mostly selected based on how easy they would be to verify based on the knowledge and skill of the researcher. While in a project with more resources, larger, more realistic, and more diverse programs could be selected for a more accurate and comprehensive end result. Similarly, the selection of languages was mostly based on how accessible they were, and based on their relative lack of complexity. Once again, in a project with more resources, this selection could be done more randomly. Most of all, the sample size of both languages and programs could be higher in such a project. As we will see in the conclusions section, some conclusions cannot be pronounced definitive purely because of these limitations.

3.4 Script and Measurements

After all of the source code had been collected, a script was used to automate the running of these programs in docker using WSL, meaning bash was used instead of a windows-native alternative like batch (the script was mostly written by AI, since the researcher had next to no knowledge of bash). Originally, an attempt was made to use batch, but support and available information on how to use it for these specific purposes proved to be lacking. The decision to use docker was made to minimize interference from other processes while the programs were running. Docker is also highly useful because containers can be stopped and started within seconds, which makes scripting hundreds of runs easier. For the majority of the selected languages, an official docker image for that language is available, e.g., popular languages like Python and Go have official images maintained by the docker community. Depending on the chosen languages, there will likely need to be some usage of non-official images (luckily, the docker hub contains many qualitative alternatives), e.g., for Lua.

First, the script starts a separate container for each language and initiates some extra procedures like installing the required libraries into each container. For the static languages, the script builds each program first within the container using the unique build commands for each language, combined with the path of the program. Most languages have built-in timing features, which allow for a more accurate measurement, if we were to do the timing in the script, we would be including the startup time for the runtime and other processes in our measurement. Unfortunately, Awk doesn't have built-in timing features, so instead, the script measures an average overhead time by repeatedly running an empty program and calculating a trimmed average. This average should correspond to the average time it takes all the processes Awk needs to start. When that average time is subtracted from the execution time of each Awk program, the result should be an approximation of the actual execution time (although even if this overhead was not accurate, the overhead turned out to be rather insignificant compared to the actual execution time for Awk programs).

Next, the programs are run one by one, the number of times this is done can be passed as a parameter when starting the script. Using this round-robin approach instead of running a program the specified number of times and then moving on to the next one ensures that the measurements are only minimally affected by temporary changes in processing speed, especially when the number of iterations is high. In terms of output, the script logs what it is doing in detail in a log file for easier debugging and prints the final results into a table in a CSV file. The bash script is included in the appendix of this paper.

This script should be run with a high number of iterations; the more iterations, the more accurate the results will be, and the more similar two separate runs of the script will be when compared. In this project, the script was run twice with 120 iterations, the results were sufficiently similar to be acceptable for use in analysis. A higher number of iterations would have resulted in more similarity, but it would have taken an unreasonable amount of extra iterations to produce a significantly higher similarity and therefore accuracy (unreasonable means that it would have taken too much time to run the script so many times, 120 iterations took around 16 hours, though a higher level of available resources could decrease this time dramatically).

Finally, Python was used to calculate the correlation, the details of which will be explained further along in this paper. Some graphs were also made to clarify the relationship between the variables.

4 Results

4.1 Raw Data

Table 1: Benchmark Results for 120 Runs

Program	C	Csharp	Swift	Go	Java	Rust	Kotlin	Python	Javascript	PHP	Perl	Lua	Julia	Awk
100_doors	113	895	1392	600	1907	505	1891	1375	2242	735	6701	1041	6168	4943
factorial	6	126	46	47	204	45	206	86	202	52	131	67	593	524
fibonacci_sequence	113	854	504	725	1108	198	1105	2012	2525	1468	7186	1037	6208	19097
fizzbuzz	42	1035	348	580	1817	456	1828	536	2212	593	216	748	5745	1661
reverse_a_string	22	144	207	130	399	503	402	223	478	181	532	394	1160	1508
selection_sort	175	2082	1388	1875	2950	446	2955	1727	3542	1874	1703	1103	16544	11005
sieve_of_eratosthenes	257	1646	2141	768	2210	1182	2266	2747	2790	1356	3479	1814	56522	20753

Above is the table of results after running the script with 120 iterations. As we can see, the results have a massive range, even for the same programs. This is further illustrated in these boxplots:

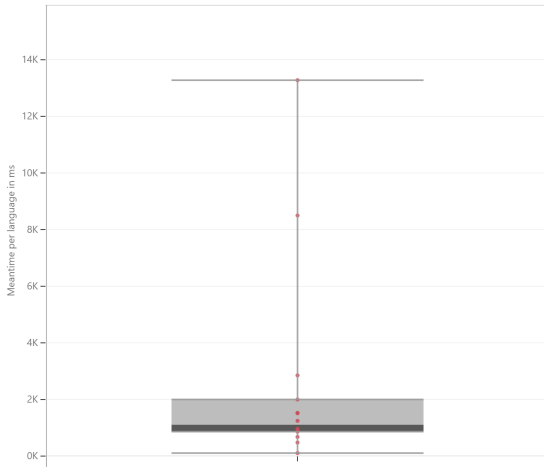

Figure 1: Boxplot for mean time of every language

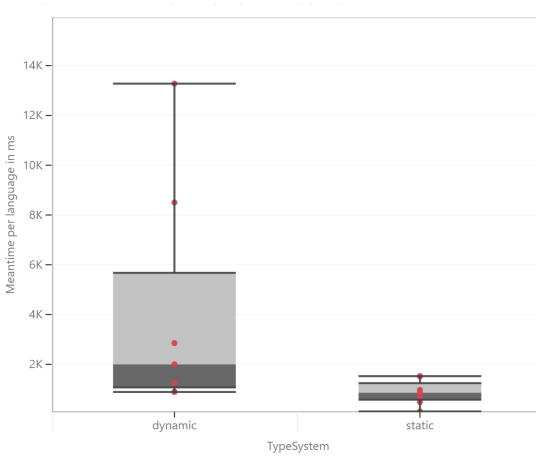

Figure 2: Boxplot for mean time of every language (static and dynamic separate)

The two main outliers we can see are Awk and Julia, both of which are dynamic, which can be seen clearly in Figure 2, the range for dynamic languages is many times larger than that of the static languages. It isn't just the range that's smaller for static languages, the median – 1999 for dynamics and 860 for statics – and the average – 4235 for dynamics and 874 for statics – are also much smaller for the static languages.

4.2 Correlation

Before effectively calculating the correlation, we have to decide how to group our data points. We have 98 data points in total, but only 14 languages, if we were to use all of these data points for our correlation, we would be artificially inflating the sample size. Instead, we will compute the mean of each language and use one data point for each language to compute the correlation.

4.2.1 Point-Biserial Correlation

Now we want to calculate the correlation, let's start by taking a simple approach, namely the Point-Biserial Correlation. This is in effect the same as a Pearson correlation, except that one of the variables is dichotomous (type system), while the other is continuous (execution time). For clarity, the Pearson correlation coefficient r is a measure of the strength and direction of the linear relationship between two quantitative variables X and Y. (Wikipedia contributors, 2025b)
This formula – for the Point-Biserial Correlation – is equivalent to calculating the Pearson Correlation with the X value coded as ones and zeros. For this calculation, static languages have been coded as 1, and dynamics as 0:

$$r_{pb} = \frac{M_1 - M_0}{s_y}\sqrt{\frac{n_1 \cdot n_0}{n^2}}$$

- M_1 is the mean of the static observations
- M_0 is the mean of the dynamic observations
- s_y is the standard deviation of all observations
- n_1 is the number of static observations
- n_0 is the number of dynamic observations
- n is the total number of observations

When we use Python to calculate this correlation, we end up with a coefficient of -0.4695 and a p-value of 0.0903. These results can be interpreted like so: A value of -0.4695 indicates a moderate negative relationship between the variables. Considering the way the binary variables have been assigned, this means that static languages tend to have lower execution times than dynamic ones. However, a p-value of 0.0903 is significantly above the standard 0.05 threshold; practically, this means that if there was no difference in the distributions of both groups, there would be a 9.03% chance that we would observe a correlation this or more extreme.

4.2.2 Rank-Biserial Correlation

Seeing as there are significant outliers, this first method of calculating the correlation was rather naive. The Point-Biserial correlation also assumes that the continuous variable is normally distributed, which is not the case here, this might also explain the high p-value. Instead, we can use the Rank-Biserial correlation, as it is much less sensitive to outliers because it is based on the ranking of data instead of raw values, it is also more robust when using such a small sample size, and it does not assume a normal distribution. The Rank-Biserial correlation is derived from the Whitney U test; which is a nonparametric test for comparing two independent samples, it ranks all observations, computes the sum of the ranks for both groups, and calculates the U statistics based on those ranks. (Wikipedia contributors, 2025a) These are the required formulas:

$$U_1 = R_1 - \frac{n_1(n_1+1)}{2},$$

$$U_2 = n_1 n_2 - U_1.$$

- n_1 is the number of static observations
- n_2 is the number of dynamic observations
- R_1 is the sum of the ranks in the static group
- R_2 is the sum of the ranks in the dynamic group (not needed for this version of the formula)

Based on these two statistics, we can derive the Rank-Biserial correlation:

$$r_{rb} = \frac{U_1 - U_2}{n_1\, n_2},$$

Once again, we can use Python to calculate this correlation. We end up with a coefficient of $\mathbf{r_{rb}}$**=-4.052**, which is reassuringly similar to the calculated Point-Biserial correlation. Just like the first method, it indicates a moderate negative relationship, meaning static languages tend to have lower execution times than dynamic ones. The Whitney U test also gave a p-value of about 0.00055, which tells us that the difference between the average execution time of each group is statistically significant (the chance that the rank differences observed are this or more extreme given that there is no difference between the distributions of both groups is 0.055%).

4.3 Per-program analysis

We can also compute this same correlation per program, which can display whether dynamic languages prevail in terms of performance in any of the programs. It can also reveal whether any individual programs are skewing the results massively in favor of static languages. Below is a table of results for the Rank-Biserial correlation per program, along with their respective effect sizes and p-values, computed using Python:

Program	p-value	Rank-Biserial	Effect Size
100 doors	0.0728	-0.5918	medium–large
factorial	0.1649	-0.4694	medium
fibonacci sequence	0.0023	-0.9184	very large
fizzbuzz	0.4557	-0.2653	small
reverse a string	0.0973	-0.5510	medium
selection sort	0.3829	-0.3061	small–medium
sieve of eratosthenes	0.0262	-0.7143	large

Table 2: Per-program Mann–Whitney p-values and signed Rank-Biserial effect sizes

The table above coincides very well with our previously observed correlation, not a single program displays a deviation in direction from our overall correlation; all of the coefficients are negative, meaning the results all suggest that static type systems tend to have better performance. We can see that there is a wide variation in effect size however, meaning that the results of some programs are closer across languages. Some of the p-values also suggest that the correlation is not at all statistically significant; like that of the selection sort program. The fibonacci-sequence program in particular shows a very large and statistically significant correlation, considering the p-value is about one-fourth of a percentage point. Overall we can see that the programs that contribute to the overall correlation the most are the sieve of eratosthenes and the fibonacci-sequence programs.

4.4 Per-language analysis

To further exemplify to what degree the pattern that statics tend to be faster than dynamics persists across languages, we can compute the mean of each language. This way, we can see whether all static languages are faster than all dynamic languages, or whether there are outliers:

Language	AvgTime (ms)
C	104.00
Rust	476.43
Go	675.00
Swift	860.86
Lua	886.29
PHP	894.14
C#	968.86
Python	1243.71
Java	1513.57
Kotlin	1521.86
JavaScript	1998.71
Perl	2849.71
Awk	8498.71
Julia	13277.14

Table 3: Average execution time by language, sorted from fastest to slowest.

Here are those results displayed in a bar chart:

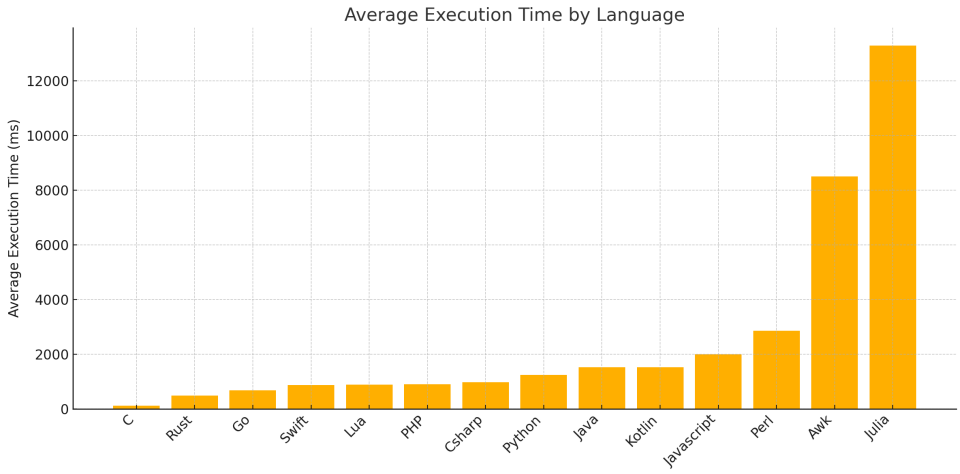

Figure 3: Average execution time by language, sorted from fastest to slowest.

As we can see, not every static language is faster than every dynamic language, although there is still a general trend in that direction, even when ignoring Awk and Julia. Both Lua and PHP are among the fastest languages, while Java and Kotlin are some of the slowest. So out of the 14 languages, 4 of them are outliers in the general trend of static being faster than dynamic. We can also see a rather massive level of performance spread, the fastest language (C at 104ms) is about 128 times faster than the slowest-performing language (Julia at 13277ms). Overall however, this chart supports the observed trend even further, even when ignoring our two main outliers.

5 Conclusion

5.1 Interpretation of Results

Based on the obtained results in the previous section, we can conclude that there is a moderate negative correlation between type system and performance, meaning that static languages tend to perform better.

Something else we can learn from the results is the effect of the genre of program on the performance. As we can see in the table of correlations per program, the sieve and fibonacci-sequence tasks display a much more significant correlation between type system and performance, while tasks like fizzbuzz and selection sort display a very weak and statistically insignificant correlation. This suggests that heavier tasks, such as recursive ones (fibonacci sequence), tend to perform better in static type systems than tasks using simple loop structures and I/O operations.

5.2 Evaluation of Criticisms

The conclusions we drew in the previous subsection can be called into question based on a few criticisms; the first being the small sample size. Even though we had 98 individual data points, to prevent artificial inflation of the sample size (which could skew the correlation calculations), we used the mean of the 14 languages. As we can see in the boxplots shown previously, there were major outliers present. Both of these outliers were dynamic, but it is entirely possible that this observation is coincidental, and that if different languages had been chosen, we would have obtained a very different distribution of outliers. Although this criticism is valid to some extent, this does not mean that the obtained correlation is invalid; the method used to compute the overall correlation – the Rank-Biserial correlation – was based on rank, not raw value, which partially mitigates the effect of those outliers, but the relatively small sample size of languages (14) allows for these outliers to remain significant. What also dampens this criticism somewhat is the fact that when looking at the mean times for each language, we can still clearly see the same pattern, even when ignoring those two dynamic outliers (although not every static language is faster than every dynamic one).

Another valid criticism is that the idea that 'heavier' programs, such as recursion-heavy ones, exemplify this correlation further, is based on the results of only 7 programs. This observed pattern can therefore not be declared definitive.

Some other, more general criticisms are that, as stated earlier, the programs and languages were not selected randomly and that the sample size simply isn't high enough, making it difficult to state any conclusions definitively.

5.3 Additional Considerations

To be clear, an observed correlation between static typing and better performance does not entail that static typing causes better performance. There are many other factors which might generally coincide with static typing that cause better overall performance.

This research also ignores certain factors about each language, such as the compilation stage for the compiled languages and the startup time for the compilers and interpreters; since we only measured the execution time, compilation was cut off completely from the measurements. If a language has horrible compilation speeds but great execution speeds, it would perform incredibly well in this type of research compared to a language with fast compilation speeds but slow execution speeds, i.e., even if the first language took 10 times longer after pressing the start button, it might still get a better result (even though, practically, this would only take place when, for example, creating code in an editor; most of the time, programs will already have been compiled). But overall, execution time is still the most logical metric for performance.

Another consideration we should make is the simplicity of these example programs compared to real-world production code. Real-world, large-scale code could give us a radically different result; e.g., real-world programs often integrate a vast array of libraries, or they might use concurrency, which none of the implementations used here do. Purely this correlation is also no fully-fledged reason to pick static type systems over dynamic ones for practical use cases, there are many other considerations to make, like ease of use, complexity, and versatility.

The consensus among computer scientists seems to be that dynamic type systems are generally more useful for prototyping, while programs written in static languages take more time and effort, but are better for large-scale, long-term projects. This paper supports that claim, at least in terms of the idea that statically typed languages display better performance and therefore have an advantage over dynamically typed languages in usefulness for large-scale projects.

Overall, the conclusions we can draw are that the results of this paper suggest that there is a moderate correlation between type system and performance, that static languages tend to perform better, and that heavier tasks, e.g., recursive programs exemplify this correlation even further. But this conclusion is not a final verdict and more expansive and fleshed-out research could provide a more definitive result.

5.4 Incentive for Future Research

Considering the limitations of this paper, there is great incentive for a successive research project following a similar structure. A follow-up project like this should have far more resources at its disposal to achieve significantly independent and worthwhile results. With these resources, the selection of languages could not only be much larger, but also random, which should achieve a much more statistically significant result. Bias caused by interfering processes could be minimized by running the programs on a 'clean-slate' machine and the amount of iterations and programs could be scaled up with a more powerful processor (along with the program complexity). Using more programs could also give a clearer view on the observed correlation between the 'heaviness' (such as recursivity) of a program and the tendency of static languages to perform (even) better.

References

GeeksForGeeks. (2019). *Type systems:dynamic typing, static typing & duck typing.* (https://www.geeksforgeeks.org/type-systemsdynamic-typing-static-typing-duck-typing)

Pierce, B. C. (2002). *Types and programming languages*. Cambridge, MA: MIT Press.

Wikipedia contributors. (2025a). *Mann–whitney u test.* https://en.wikipedia.org/wiki/Mann-Whitney_U_test.

Wikipedia contributors. (2025b). *Pearson correlation coefficient.* https://en.wikipedia.org/wiki/Pearson_correlation_coefficient.

6 Appendix

A Bash script

```
#!/bin/bash

# Usage: ./benchmark.sh [number_of_runs]
# Defaults to 30 if not provided.
runs=${1:-30}

echo "Starting benchmark with $runs iterations..."

# Define the order of languages (this will also determine CSV column order)
    .
langOrder=("C" "Csharp" "Swift" "Go" "Java" "Rust" "Kotlin" "Python" "
    Javascript" "PHP" "Perl" "Lua" "Julia" "Awk")

# Define Docker images for each language.
declare -A images=(
["C"]="gcc"
["Csharp"]="mcr.microsoft.com/dotnet/sdk"
["Swift"]="swift"
["Go"]="golang"
["Java"]="openjdk:11"
["Rust"]="rust"
["Kotlin"]="openjdk:11"
["Python"]="python"
["Javascript"]="node"
["PHP"]="php"
["Perl"]="perl"
["Lua"]="nickblah/lua"
["Julia"]="julia"
["Awk"]="busybox"
)

# Define file extensions for each language.
declare -A exts=(
["C"]=".c"
["Csharp"]=".cs"
["Swift"]=".swift"
["Go"]=".go"
["Java"]=".java"
["Rust"]=".rs"
["Kotlin"]=".kt"
["Python"]=".py"
["Javascript"]=".js"
["PHP"]=".php"
["Perl"]=".pl"
["Lua"]=".lua"
["Julia"]=".jl"
["Awk"]=".awk"
)

# For interpreted languages, define a runner command using placeholders.
declare -A runners=(
["Python"]="python /app/{prog}{ext}"
["Javascript"]="node /app/{prog}{ext}"
["PHP"]="php /app/{prog}{ext}"
["Perl"]="perl /app/{prog}{ext}"
["Lua"]="LUA_PATH='/usr/local/share/lua/5.4/?.lua;/usr/local/share/lua
    /5.4/?/init.lua;./?.lua' LUA_CPATH='/usr/local/lib/lua/5.4/?.so;./?.so'
```

```
    lua /app/{prog}{ext}"
["Julia"]="julia /app/{prog}{ext}"
["Awk"]="sed 's/\\r//g' /app/{prog}{ext} | awk -f -"
)

# Define which languages require a build step.
declare -A buildRequired=(
["C"]="yes"
["Csharp"]="yes"
["Swift"]="yes"
["Java"]="yes"
["Rust"]="yes"
["Kotlin"]="yes"
["Go"]="yes"
)

# Define build command templates for languages that require compilation.
declare -A builds=(
["C"]="gcc /app/{prog}.c -o /app/{prog}_exe"
["Csharp"]="dotnet new console --force --name {prog} -o /tmp/tempproj_{prog
    } && cp /app/{prog}.cs /tmp/tempproj_{prog}/Program.cs && dotnet build /
    tmp/tempproj_{prog} -c Release -o /app && rm -rf /tmp/tempproj_{prog}"
["Swift"]="swiftc /app/{prog}.swift -o /app/{prog}_exe"
["Java"]="javac /app/{prog}.java"
["Rust"]="rustc /app/{prog}.rs -o /app/{prog}_exe"
["Kotlin"]="apt-get update -qq && apt-get install -y -qq wget unzip && wget
     -q https://github.com/JetBrains/kotlin/releases/download/v1.6.21/kotlin
    -compiler-1.6.21.zip -O kotlin.zip && rm -rf /tmp/kotlinc && unzip -qq -
    o kotlin.zip -d /tmp && /tmp/kotlinc/bin/kotlinc /app/{prog}.kt -include
    -runtime -d /app/{prog}.jar"
["Go"]="go build -o /app/{prog}_exe /app/{prog}.go"
)

# Define run command templates for built (compiled) languages.
declare -A runsCmd=(
["C"]="/app/{prog}_exe"
["Csharp"]="dotnet /app/{prog}.dll"
["Swift"]="/app/{prog}_exe"
["Java"]="java -cp /app {prog}"
["Rust"]="/app/{prog}_exe"
["Kotlin"]="java -jar /app/{prog}.jar"
["Go"]="/app/{prog}_exe"
)

# List of programs (source file names without extension).
programs=("100_doors" "factorial" "fibonacci_sequence" "fizzbuzz" "
    reverse_a_string" "selection_sort" "sieve_of_eratosthenes")

# Set log and CSV files.
logFile="benchmark_results.txt"
csvFile="benchmark_table.csv"

# Create or clear the log file.
echo "Benchmark Results" > "$logFile"
echo "==================" >> "$logFile"

# Initialize associative arrays to store total times and average times.
declare -A totalTimes
declare -A avgTimes
for prog in "${programs[@]}"; do
for lang in "${langOrder[@]}"; do
totalTimes["$prog,$lang"]=0
done
```

```
done

# Initialize an associative array to store measured overhead for each
    language.
declare -A overheadTimes

echo "Starting containers and building programs..."
# --- Start persistent containers, build all programs, and measure overhead
     ---
for lang in "${langOrder[@]}"; do
folder="$lang"
image="${images[$lang]}"
containerName="benchmark_${lang}"

echo "Starting container for $lang..."
docker run -d --name "$containerName" -v "$(pwd)/programs/$folder":/app "
    $image" tail -f /dev/null >> "$logFile" 2>&1

# For Lua, install luarocks and LuaSocket for Lua 5.4.
if [ "$lang" == "Lua" ]; then
echo "Installing luarocks and LuaSocket in Lua container..."
docker exec "$containerName" sh -c 'if command -v apt-get >/dev/null 2>&1;
    then apt-get update && apt-get install -y luarocks; elif command -v apk
    >/dev/null 2>&1; then apk update && apk add luarocks; else echo "No
    suitable package manager found"; fi && luarocks install luasocket --lua-
    version=5.4' >> "$logFile" 2>&1
fi

# Build phase (if required).
if [ "${buildRequired[$lang]}" == "yes" ]; then
for prog in "${programs[@]}"; do
if [[ "$lang" == "Java" && "$prog" == "100_doors" ]]; then
progName="OneHundredDoors"
else
progName="$prog"
fi
buildCmdTemplate="${builds[$lang]}"
buildCmd="${buildCmdTemplate//\{prog\}/$progName}"
echo "Building $progName for $lang..."
docker exec "$containerName" sh -c "$buildCmd" >> "$logFile" 2>&1
build_ret=$?
if [ $build_ret -ne 0 ]; then
echo "Error: Building $progName for $lang failed with exit code $build_ret"
     | tee -a "$logFile"
fi
done
fi

# For non-Awk languages, set overhead to 0.
if [ "$lang" != "Awk" ]; then
overheadTimes["$lang"]=0
echo "Overhead for $lang set to 0."
else
echo "Measuring overhead for Awk..."
num_overhead=50
overheads=()
for ((i=1; i<=num_overhead; i++)); do
start_overhead=$(date +%s%N)
docker exec "$containerName" sh -c "true" >> /dev/null 2>&1
end_overhead=$(date +%s%N)
overhead_run=$(( (end_overhead - start_overhead) / 1000000 ))
overheads+=("$overhead_run")
echo "Overhead measurement $i for Awk: ${overhead_run} ms" >> "$logFile"
```

```
done
sorted=$(printf "%s\n" "${overheads[@]}" | sort -n)
declare -a sorted_overheads
while IFS= read -r line; do
sorted_overheads+=("$line")
done <<< "$sorted"
trim_start=5
trim_end=5
sum=0
count=0
for ((j=trim_start; j < num_overhead - trim_end; j++)); do
sum=$(( sum + sorted_overheads[j] ))
count=$(( count + 1 ))
done
overhead_avg=$(( sum / count ))
overheadTimes["$lang"]=$overhead_avg
echo "Measured overhead for Awk (trimmed average): ${overhead_avg} ms"
echo "Overhead for Awk: ${overhead_avg} ms" >> "$logFile"
fi
done

echo "Starting round-robin execution..."
# --- Round-robin execution: For each iteration, run every program in every
    language ---
for ((iter=1; iter<=runs; iter++)); do
echo "Iteration $iter of $runs..."
echo "Iteration $iter" >> "$logFile"
for lang in "${langOrder[@]}"; do
containerName="benchmark_${lang}"
ext="${exts[$lang]}"
overhead_avg=${overheadTimes["$lang"]}
for prog in "${programs[@]}"; do
if [[ "$lang" == "Java" && "$prog" == "100_doors" ]]; then
progName="OneHundredDoors"
else
progName="$prog"
fi
if [ "${buildRequired[$lang]}" == "yes" ]; then
runCmdTemplate="${runsCmd[$lang]}"
runCmd="${runCmdTemplate//\{prog\}/$progName}"
else
runner="${runners[$lang]}"
runCmd="${runner//\{prog\}/$prog}"
runCmd="${runCmd//\{ext\}/$ext}"
fi

echo "Running $progName in $lang..."
if [ "$lang" == "Awk" ]; then
start=$(date +%s%N)
docker exec "$containerName" sh -c "$runCmd" >> /dev/null 2>&1
ret=$?
end=$(date +%s%N)
raw_elapsed=$(( (end - start) / 1000000 ))
if [ $raw_elapsed -gt $overhead_avg ]; then
elapsed=$(( raw_elapsed - overhead_avg ))
else
elapsed=0
fi
if [ $ret -ne 0 ]; then
echo "Warning: $progName in $lang failed (exit code $ret)" | tee -a "
    $logFile"
fi
else
```

```
output=$(docker exec "$containerName" sh -c "$runCmd" 2>&1)
ret=$?
elapsed=$(echo "$output" | tail -n 1)
echo "Final elapsed time for $progName in $lang: $elapsed"
if ! [[ "$elapsed" =~ ^[0-9]+(.[0-9]+)?$ ]]; then
elapsed=0
fi
if [ $ret -ne 0 ]; then
echo "Warning: $progName in $lang failed (exit code $ret)" | tee -a "
    $logFile"
fi
fi
elapsed=$(printf "%.0f" "$elapsed")
totalTimes["$prog,$lang"]=$(( totalTimes["$prog,$lang"] + elapsed ))
done
done
done

echo "Calculating averages and logging results..."
for prog in "${programs[@]}"; do
for lang in "${langOrder[@]}"; do
avgTimes["$prog,$lang"]=$(( totalTimes["$prog,$lang"] / runs ))
echo "Program: $prog in $lang -- Average Execution Time: ${avgTimes["$prog,
    $lang"]} ms" >> "$logFile"
done
done

echo "Stopping and removing containers..."
for lang in "${langOrder[@]}"; do
containerName="benchmark_${lang}"
docker stop "$containerName" > /dev/null 2>&1 && docker rm "$containerName"
     > /dev/null 2>&1
done

echo "Producing CSV output..."
{
    printf "Program"
    for lang in "${langOrder[@]}"; do
    printf ",%s" "$lang"
    done
    printf "\n"
    for prog in "${programs[@]}"; do
    printf "%s" "$prog"
    for lang in "${langOrder[@]}"; do
    key="$prog,$lang"
    value="${avgTimes[$key]}"
    if [ -z "$value" ]; then
    value="NA"
    fi
    printf ",%s" "$value"
    done
    printf "\n"
    done
} > "$csvFile"

echo "Benchmarks complete."
echo "Detailed log available in $logFile and CSV output in $csvFile."
```

B C# Benchmark Source Code

100_doors.cs

```
using System;
using System.Diagnostics;

class Program
{
    static void Main()
    {
        Stopwatch sw = Stopwatch.StartNew();

        for (int iterations = 0; iterations < 10000; iterations++)
        {
            bool[] doors = new bool[100];  // Array of 100 doors,
                all initially closed

            // Loop through the passes
            for (int i = 1; i <= 100; i++)
            {
                for (int j = i; j <= 100; j += i)
                {
                    doors[j - 1] = !doors[j - 1];  // Toggle door
                        state (0-based index)
                }
            }

            // Print the door states
            for (int i = 0; i < 100; i++)
            {
                if (doors[i])
                {
                    Console.WriteLine($"Door {i + 1} is open.");
                }
                else
                {
                    Console.WriteLine($"Door {i + 1} is closed.");
                }
            }
        }
        sw.Stop();
        // Output only the total elapsed time in milliseconds.
        Console.Write(sw.Elapsed.TotalMilliseconds);
    }
}
```

factorial.cs

```
using System;
using System.Diagnostics;

class Program
{
```

```
    static int Factorial(int number)
    {
        var accumulator = 1;
        for (var factor = 1; factor <= number; factor++)
        {
            accumulator *= factor;
        }
        return accumulator;
    }

    static void Main()
    {
        Stopwatch sw = Stopwatch.StartNew();
        for (int iterations = 0; iterations < 100000; iterations++)
        {
            Console.WriteLine(Factorial(10));
        }
        sw.Stop();
        Console.Write(sw.Elapsed.TotalMilliseconds);
    }
}
```

fibonacci_sequence.cs

```
using System;
using System.Diagnostics;

class FibonacciGenerator
{
    static void Main()
    {
        Stopwatch sw = Stopwatch.StartNew();
        for (int iterations = 0; iterations < 100000; iterations++)
        {
            int count = 10; // Number of Fibonacci numbers to
               generate
            for (int i = 0; i < count; i++)
            {
                Console.Write(Fibonacci(i) + " ");
            }
            Console.WriteLine();
        }
        sw.Stop();
        Console.Write(sw.Elapsed.TotalMilliseconds);
    }

    static int Fibonacci(int n)
    {
        if (n <= 0) return 0;
        if (n == 1) return 1;
        return Fibonacci(n - 1) + Fibonacci(n - 2); // Recursive
           calculation
    }
}
```

fizzbuzz.cs

```
using System;
using System.Diagnostics;
class Program
{
    static void Main(string[] args)
    {
        Stopwatch sw = Stopwatch.StartNew();
        for (int iterations = 0; iterations < 10000; iterations++)
        {
            for (int i = 1; i <= 100; i++)
            {
                if (i % 15 == 0)
                {
                    Console.WriteLine("FizzBuzz");
                }
                else if (i % 3 == 0)
                {
                    Console.WriteLine("Fizz");
                }
                else if (i % 5 == 0)
                {
                    Console.WriteLine("Buzz");
                }
                else
                {
                    Console.WriteLine(i);
                }
            }
        }
        sw.Stop();
        Console.Write(sw.Elapsed.TotalMilliseconds);
    }
}
```

reverse_a_string.cs

```
using System;
using System.Diagnostics;

class StringReverser
{
    static void Main()
    {
        Stopwatch sw = Stopwatch.StartNew();
        for (int iterations = 0; iterations < 100000; iterations++)
        {
            string original = "Hello, World!"; // Hardcoded string
            string reversed = ReverseString(original);

            Console.WriteLine(original);
            Console.WriteLine(reversed);
        }
        sw.Stop();
```

```
            Console.Write(sw.Elapsed.TotalMilliseconds);
        }

        static string ReverseString(string input)
        {
            char[] reversedArray = new char[input.Length];

            // Manually reverse the string
            for (int i = 0; i < input.Length; i++)
            {
                reversedArray[input.Length - 1 - i] = input[i];
            }

            return new string(reversedArray);
        }
    }
```

selection_sort.cs

```
using System;
using System.Diagnostics;

class SelectionSortExample
{
    static void Main(string[] args)
    {
        Stopwatch sw = Stopwatch.StartNew();
        for (int iterations = 0; iterations < 100000; iterations++)
        {
            int[] arr = { 29, 10, 14, 37, 13, 54, 76, 98, 12, 1,
                64, 13, 512 };

            Console.WriteLine("Original array:");
            PrintArray(arr);

            SelectionSort(arr);

            Console.WriteLine("Sorted array:");
            PrintArray(arr);
        }
        sw.Stop();
        Console.Write(sw.Elapsed.TotalMilliseconds);
    }

    static void SelectionSort(int[] arr)
    {
        int n = arr.Length;
        for (int i = 0; i < n - 1; i++)
        {
            int minIndex = i;
            for (int j = i + 1; j < n; j++)
            {
                if (arr[j] < arr[minIndex])
                {
                    minIndex = j;
                }
```

```
                }
                if (minIndex != i)
                {
                    int temp = arr[i];
                    arr[i] = arr[minIndex];
                    arr[minIndex] = temp;
                }
            }
        }

        static void PrintArray(int[] arr)
        {
            foreach (int num in arr)
            {
                Console.Write(num + " ");
            }
            Console.WriteLine();
        }
    }
```

sieve_of_eratosthenes.cs

```
using System;
using System.Diagnostics;

class SieveOfEratosthenes
{
    static void Main()
    {
        Stopwatch sw = Stopwatch.StartNew();
        for (int iterations = 0; iterations < 1000; iterations++)
        {
            int maxPrime = 10000; // Hardcoded limit
            bool[] primes = GeneratePrimes(maxPrime);

            // Print prime numbers
            for (int i = 2; i <= maxPrime; i++)
            {
                if (primes[i])
                {
                    Console.WriteLine(i);
                }
            }
        }
        sw.Stop();
        Console.Write(sw.Elapsed.TotalMilliseconds);
    }

    static bool[] GeneratePrimes(int maxPrime)
    {
        bool[] isPrime = new bool[maxPrime + 1];

        // Assume all numbers are prime initially
        for (int i = 2; i <= maxPrime; i++)
        {
            isPrime[i] = true;
```

```
        }

        // Mark non-prime numbers
        for (int i = 2; i <= maxPrime; i++)
        {
            if (isPrime[i])
            {
                for (int j = i * i; j <= maxPrime; j += i)
                {
                    isPrime[j] = false;
                }
            }
        }
        return isPrime;
    }
}
```