Computing for Data Science

Assignment 1

Prof. Aaron

Nisarg Shah – UID No. 2509005

ABSTRACT

This study evaluates the performance, memory utilization, and development tradeoffs for four programming languages, namely Python, Java, Go, and Rust, by implementing a recursive Fibonacci algorithm to compute the 40th Fibonacci number (assuming it starts from 0 and 1). By comparing the output of the code in all the languages using different performance metrics, we can conclude that Python is the slowest language, and Rust is the fastest. Moreover, Go is the most memory efficient, whereas Python is the least.

OBJECTIVE

The objective of this study is to conduct a comparative analysis of Python and three other modern programming languages by implementing a common algorithm and evaluating their performance across multiple metrics. The analysis focuses on execution speed, computational reliability, and resource efficiency, while also examining the tradeoffs between development and performance.

- **Languages Selected:** Python, Java, Go, Rust
- **Algorithm Chosen:** Recursive Fibonacci for $n = 40$, starting from 0 and 1
- **Metrics Evaluated:** CPU Time, Execution Time, Memory Used, Stability Under Load, Ease of Debugging

METHOD

The following steps were carried out to conduct the comparative performance analysis:

1. The pseudocode of the recursive Fibonacci algorithm is as follows:

Algorithm 1 Recursive Fibonacci

```
Take n = 40
if n = 1 then
    F_n = 0
else if n = 2 then
    F_n = 1
else
    F_n = F_{n-1} + F_{n-2}
```

2. Implemented the above algorithm separately in Python, Java, Go, and Rust using equivalent logic.
3. Executed each implementation under similar system and environment conditions (on VS Code) to ensure fairness.
4. Measured/observed the following performance metrics for each run:
    - CPU Time
    - Execution Time
    - Memory Used
    - Stability Under Load
    - Ease of Debugging
5. Recorded the observed values and compared results to determine the fastest and slowest languages, as well as their relative reliability and efficiency.
6. Analyzed tradeoffs between development time and runtime performance.

OBSERVATIONS

I have attached code for each language in the repository.
The following pictures are screenshots of the output of the algorithm in each language, along with a table compiling them.

```
@nisarg-m-shah →/workspaces/Computing-for-Data-Science-Assignment-1 (main) $ python3 Fibonacci.py
Language: Python
Fibonacci(40) = 63245986
Execution time: 11.225462 seconds
CPU time: 11.131397 seconds
Memory used: 9984 KB
```

Python Output for Fibonacci($n = 40$)

```
@nisarg-m-shah →/workspaces/Computing-for-Data-Science-Assignment-1 (main) $ java Fibonacci
Language : Java
Fibonacci(40) = 63245986
Execution time: 0.286531 seconds
CPU time: 0.283247 seconds
Memory used: 1934 KB
```

Java Output for Fibonacci($n = 40$)

```
@nisarg-m-shah →/workspaces/Computing-for-Data-Science-Assignment-1 (main) $ go run Fibonacci.go
Language: Golang
Fibonacci(40) = 63245986
Execution time: 0.451503 seconds
CPU time: 0.440000 seconds
Memory used: 4.02 KB
```

Go Output for Fibonacci($n = 40$)

```
@nisarg-m-shah →/workspaces/Computing-for-Data-Science-Assignment-1/Fibonacci (main) $ cargo run --release
    Finished `release` profile [optimized] target(s) in 0.01s
     Running `target/release/Fibonacci`
Language : Rust
Fibonacci(40) = 63245986
Execution time (wall): 0.215277 seconds
CPU time: 0.213508 seconds
Memory used: 2048 KB
```

Rust Output for Fibonacci($n = 40$)

Language	Execution Time (s)	CPU Time (s)	Memory (KB)	Stability	Ease of Debugging
Python	11.2254	11.1313	9984	Unstable	Easy
Java	0.2865	0.2832	1934	Stable	Medium
Go	0.4515	0.4400	4.02	Stable	Intermediate
Rust	0.2152	0.2135	2048	Stable	Hard

Performance metrics for Fibonacci(40) across four programming languages

The overlying table shows the results for **CPU Time**, **Execution Time**, **Memory Usage**, **Stability Under Load**, and **Ease of Debugging** for each language. All results are from running the recursive Fibonacci program with $n = 40$ on the same machine and under similar conditions.

Even though we tried to measure each language in the same way, there were some challenges. For Java and Go, most of the memory was used on the stack, but many profiling tools only report heap memory. This made Go's reported memory usage extremely low. For Java, we were able to get a more accurate number, but for Go it is still likely lower than the real usage. This also made it harder to compare memory results fairly.

RESULTS

- **CPU and Execution Time:** In terms of CPU time, Rust (when run with `cargo run -release`) was the fastest, taking about 0.2152 seconds. Java came next at 0.2865 seconds, followed by Go at 0.4515 seconds. Python was much slower, taking over 11 seconds. In this study, CPU time trends were consistent with execution time results. While Execution time reflects the total elapsed time from start to finish, CPU time provides a more reliable measure of the language's computational efficiency, as it is less affected by external factors. Thus we shall not delve into execution time, as it is already confirmed that it follows the same pattern.
- **Memory Usage:** Rust and Java used about 2 MB each. Python used the most memory at almost 10 MB. Go's reported memory usage was just 4.02 KB, but this is likely an undercount because it doesn't include stack memory.
- **Stability Under Load:** The Fibonacci algorithm takes more time as n increases. Python already takes over 10 seconds for $n = 40$ and fails to return a result quickly for $n = 50$. Rust, Java, and Go all computed $n = 50$ within 1-2 seconds. Rust runs slower in debug mode (`cargo run`)

because it rebuilds the program every time, but is fastest when optimised.

- **Ease of Debugging** Python was the easiest to debug, with clear error messages and very little setup. Java was also manageable, but its error messages can be quite long. Go was fairly easy, but understanding its memory use was tricky. Rust was the hardest to debug ; you must set up a project folder, place your code in `src/main.rs`, and use `cargo run` to execute. It also shows a lot of extra messages every time you run it, which can get in the way.

Fastest, Slowest, and Best Memory Usage

- **Fastest:** Rust (optimised mode)
- **Slowest:** Python
- **Best Memory Efficiency:** Rust (real measurement), though Go appears lowest because of incomplete memory reporting.

Trade-Offs

There is a clear trade-off between how fast the program runs and how easy it is to write:

- Python is quick to write and debug, but very slow and memory-heavy.
- Java and Go give good performance without being too hard to write.
- Rust is very fast and memory-efficient, but harder to learn and slower to develop in.

Conclusion

If performance is the top priority, Rust is the best choice despite the extra work. For quick and simple coding, Python is the easiest. Java and Go sit in the middle, offering a balance of speed and ease of use.