Mahwish Dadan | Roll no. 16

Title: "Comparative Study of Programming Languages for Stream-Based Data Processing"

1. **Introduction**
   This report presents a comparative benchmarking analysis of four widely-used programming languages: Python, Java, Go, and Rust. Each language offers distinct strengths — Python for its simplicity and rich ecosystem, Java for its platform independence, Go for its fast compilation and concurrency model, and Rust for its memory safety and high performance.

   By running the same set of tasks across all four, this report evaluates their performance in terms of execution time, memory usage, binary size, and code complexity, offering insights into their relative strengths, trade-offs, and suitability for different types of applications.

2. **Languages Chosen**

   The following four languages were selected for benchmarking due to their relevance in modern software and data systems:

   - Python – Interpreted, high-level, and easy to use.
   - Java – Runs on the JVM, platform independent and widely used in enterprise environments.
   - Go – Compiled language designed for fast performance and simplicity.
   - Rust – Systems-level compiled language known for memory safety and speed.

3. **Task Description**
   Five tasks were selected to benchmark different performance aspects of each language, including CPU usage, memory efficiency, and concurrency. These tasks are designed to simulate real-world scenarios relevant to data processing and systems development.

   - **TASK 1: File Compression**

     This task involved compressing a 10MB text file using each language’s standard gzip or equivalent library. It was designed to test file I/O performance, CPU usage during compression, and memory management. This simulates scenarios where real-time data streams need to be compressed before storage or transmission.

   - **TASK 2: Line-by-Line Word Count**

     This task involved reading a 10MB text file line by line and counting how many lines contained the word "data". To simulate realistic log processing, the input file was generated using a simple Python script. It randomly wrote lines either containing or not containing the target word, with approximately 30% of lines including the word "data". This allowed for a balanced test of string matching and conditional logic. This assesses buffered I/O, string processing efficiency, and control flow handling. It represents typical use cases in log processing, text mining, or ETL pipelines.

   - **TASK 3: Sorting and Binary Search**

     A list of one million randomly generated integers was first sorted using the language’s built-in sorting function. A binary search was then performed for 1,000 randomly selected values. This task evaluates algorithmic performance, memory layout handling, and CPU-bound computation, which are common in analytics and database operations.

   - **TASK 4: Matrix Multiplication**

Two 500×500 matrices containing random integers were multiplied using a naïve triple-nested loop method. This task stresses numerical computation and memory access efficiency. It serves as a benchmark for scientific computing, simulations, and machine learning workloads.

- **TASK 5: Multithreaded File Processing**

  The final task extended the word count logic by dividing the file into chunks and processing them in parallel threads. Each thread counted matching lines independently before aggregating the results. This tested concurrency primitives, thread scaling, and real-world parallel performance — critical in large-scale data systems and real-time analytics.

4. **Metrics Measured**

The following metrics were used to evaluate the languages:

- **Execution Time (ms):**
  Total time taken to complete the task, measured in milliseconds. This reflects raw performance and efficiency of the language runtime or compiled output.

- **Peak Memory Usage (MB):**
  Maximum memory consumed during execution. This helps assess how efficiently each language manages memory under load.

- **CPU Usage (%):**
  Average CPU utilization during task execution, giving insight into how computationally intensive the task was and how well each language utilizes system resources.

- **Binary Size (bytes):**
  For compiled languages (Java, Go, Rust), the final binary or executable size was recorded. This can impact deployment, especially in constrained environments

- **Lines of Code (LOC):**
  Approximate number of lines written to implement each task. Fewer lines generally indicate better expressiveness and maintainability.

- **Difficulty of Coding (Subjective):**
  A subjective measure based on development effort, ease of debugging, availability of libraries, and clarity of language syntax. Ranked qualitatively as *Easy*, *Moderate*, or *Hard*.

5. **Benchmark Setup**

All benchmarks were conducted on a personal computer with the following specifications:

- **Hardware:** Ryzen 4000 series CPU, 16GB RAM
- **Operating System:** Windows 11 (64-bit)

The software versions used were:

- **Python:** 3.10.0
- **Java:** 11.0.21 (LTS)
- **Go:** 1.24.6 (windows/amd64)
- **Rust (Cargo):** 1.89.0

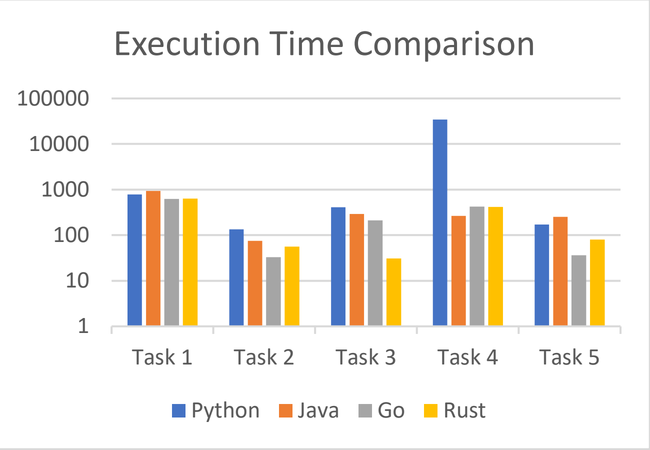
For each task, execution time was measured programmatically within the code to ensure accurate timing results. CPU usage and peak memory consumption (private bytes) were monitored using **Process Explorer** on Windows.

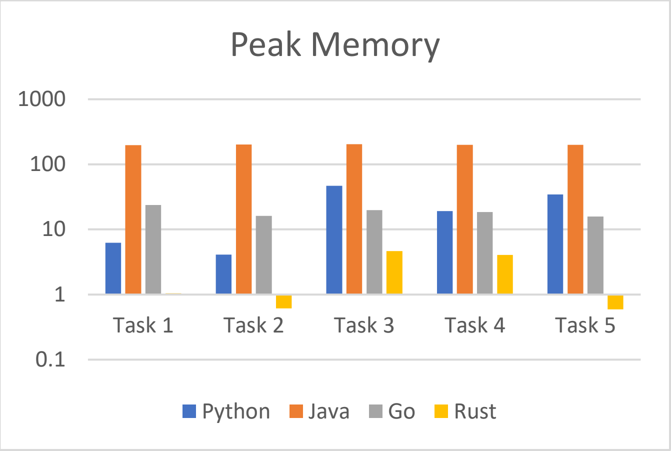
Binary sizes for compiled languages were retrieved via Windows PowerShell commands, for example: (Get-Item .\Task1.exe).Length

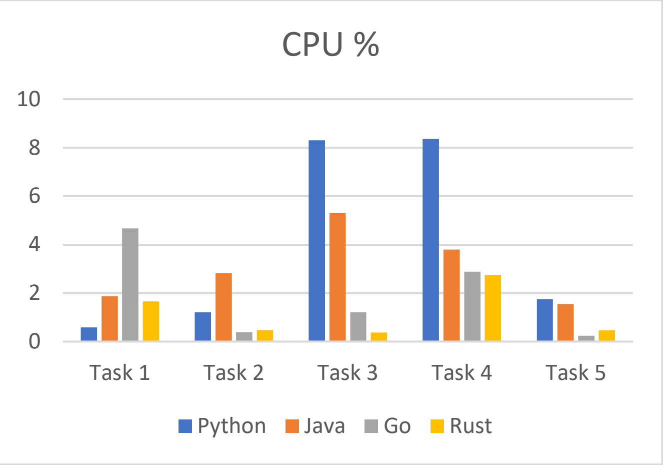
Lines of code (LOC) were counted using PowerShell's line count command, for example: (Get-Content .\Task1.java| Measure-Object -Line).Lines

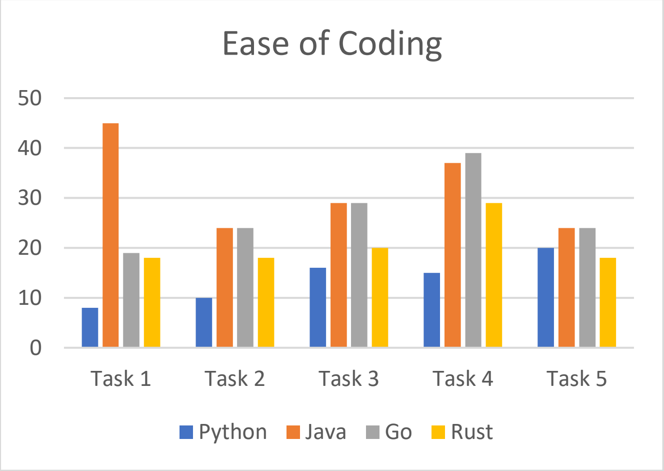
6. **Results**

	Languages	**Execution time (ms)**	**Peak memory (mb)**	**CPU usage (%)**	**Binary size (bytes)**	**Lines of code**	**Ease of coding**
TASK 1	**Python**	787	6.21	0.59	NA	8	4.5
	Java	938	197	1.87	1798	45	3
	Go	627	23.78	4.67	2645504	19	4
	Rust	632	1.04	1.66	215040	18	2
TASK 2	**Python**	134.6	4.117	1.21	NA	10	5
	Java	75	201.9	2.82	1903	24	3.5
	Go	33	16.15	0.39	2362880	24	4
	Rust	56.48	0.614	0.48	154112	18	3
TASK 3	**Python**	412.3	47	8.30	NA	16	4.5
	Java	292	203	5.30	1517	29	3
	Go	210	19.91	1.20	2364416	29	3.5
	Rust	31	4.66	0.37	168960	20	3.5
TASK 4	**Python**	34670.90	19.05	8.35	NA	15	4
	Java	265	199.45	3.79	1747	37	2.5
	Go	424	18.5	2.89	2354688	39	3.5
	Rust	419	4.07	2.75	168448	29	3
TASK 5	**Python**	171	34.53	1.75	NA	20	4.5
	Java	252	199.10	1.55	3145	24	3
	Go	36	15.87	0.24	2362880	24	4
	Rust	80.52	0.59	0.47	154112	18	3









7. **Interpretation**

- **Performance Trends**

  - Rust consistently outperformed other languages in execution time, especially for compute-heavy tasks like sorting and matrix multiplication.
  - Go also showed strong performance, particularly in multithreaded and file I/O tasks, thanks to its lightweight concurrency model.
  - Java was reasonably fast but suffered from very high memory usage across all tasks due to JVM overhead.
  - Python was consistently the slowest, especially in compute-bound tasks, but its code was the shortest and easiest to write.

- **Memory usage analysis**

  - Rust used the least memory across all tasks — often 10–50× less than Java and 2–5× less than Go.
  - Java had a consistently high memory footprint (~200 MB), which is expected due to the JVM.
  - Python's memory use was moderate, but spiked in tasks involving large data structures (e.g., Task 3 & 5).
  - Go maintained a balance — better than Java, but not as lightweight as Rust.

- **Binary Size and Compilation**

- Rust produced the smallest binaries, especially optimized in release mode.
- Go binaries were large due to static linking, but this enables portability.
- Java had tiny compiled .class files, but requires the JVM to run — trade-off between size and portability.
- Python is interpreted — no binaries — which simplifies development but limits raw performance.

- **Ease of Coding & Developer Productivity**
  - Python had the fewest lines of code and the highest ease of coding ratings — ideal for prototyping.
  - Go offered clean syntax, good concurrency support, and was second-easiest overall.
  - Java required more boilerplate code, lowering ease of use despite good library support.
  - Rust was more verbose and stricter, but powerful

- **Multithreading**
  - Go and Rust performed exceptionally in multithreaded workloads.
  - Python's threading was limited by the Global Interpreter Lock (GIL), resulting in modest improvements.
  - Java’s multithreading worked well, though with memory cost.

8. Conclusion

This benchmarking study compared Python, Java, Go, and Rust across five representative tasks targeting I/O, computation, and concurrency. The results highlight key trade-offs:

- **Python** is the easiest to write and understand, but it consistently underperforms in execution speed and memory usage, especially in CPU-bound tasks.

- **Java** offers stable performance and mature tooling, but suffers from high memory consumption and larger code verbosity.

- **Go** strikes a solid balance between performance and ease of development. Its lightweight binaries, fast execution, and simple concurrency model make it well-suited for scalable systems.

- **Rust** consistently delivered the **best performance** (speed and memory efficiency), especially in computation-heavy and multithreaded scenarios. However, its learning curve is higher due to strict safety guarantees and verbose syntax

Language	**Best For**
Rust	High-performance, memory-efficient applications
Go	Fast development, concurrency-heavy systems
Python	Quick scripting, prototypes, small data tools
Java	Stable enterprise apps where JVM integration is beneficial