**Name-Michael Fernandes**

Name: Michael Fernandes  
UID: 2059006  
Roll No: 06  
Title: Benchmarking Python, C, C++,Go and Java on Numerical Workloads  
Tasks:  
1. Enumerate odd integers and Fibonacci Sequence - 1M  
2. Compute N-Body Simulation  
3. Compute Deep Recursion  
Languages Used: Python, C, C++, Java,Go  
Goal: Compare execution speed, memory behavior, stability under load, debugging and development effort  
Profilers Used: cPython, VirtualVM, Valgrind, gprof,pprof

**Abstract**

This study benchmarks five widely-used programming languages—**Python**, **Java**, **C**, **C++**, and **Go**—to evaluate their performance characteristics across diverse compute-bound workloads. The selected programs include:

• **Odd number enumeration** up to 1 million  
• **Fibonacci series generation** (1 million terms)  
• **N-body gravitational simulation**  
• **Deep recursive function calls**

These workloads were chosen to stress different aspects of system behavior: loop performance, memory allocation and access patterns, floating-point arithmetic, function call overhead, and stack management.

Each implementation was designed to be algorithmically equivalent to ensure fairness. Through microbenchmarks, we expose intrinsic runtime costs such as loop overhead, allocation patterns, bounds checking, recursion limits, garbage collection pressure, and JIT compilation effects.

Our analysis focuses on **execution speed**, **memory usage**, **concurrency support**, **runtime safety**, and **profiling complexity**, highlighting trade-offs between low-level control in compiled languages and productivity in managed or memory-safe environments. This study aims to guide developers and researchers seeking language-level performance insights for CPU-intensive applications.

**1. Introduction**

Programming languages differ significantly in how they balance execution speed, memory consumption, concurrency capabilities, and developer ergonomics. We benchmark **Python, Java, C, C++, and Go** using a common suite of compute-bound workloads to compare:

• **Raw performance** (CPU time)  
• **Memory behavior** and allocation patterns  
• **Concurrency** and threading model efficiency  
• **Runtime stability** and safety guarantees  
• **Ease of development**, profiling, and observability

**Benchmark Workloads**

**1. Enumerating Odd Numbers up to 1 Million**  
• **Purpose:** Tests raw iteration performance, branch handling, and integer operations.  
• **Insights:** Highlights loop overhead and runtime dispatching costs.

**2. Computing the Fibonacci Series**  
• **Variants:** Iterative, recursive, and memoized.  
• **Purpose:** Measures recursion support, function call cost, and stack depth handling.  
• **Insights:** Exposes deep call costs and opportunities for memoization/tail-call optimization.

**3. N-body Gravitational Simulation**  
• **Purpose:** Intensive floating-point arithmetic with nested loops.  
• **Insights:** Highlights CPU throughput, floating-point performance, cache behavior, and memory access efficiency.

**4. Deep Recursive Function Calls**  
• **Purpose:** Tests recursion limit, stack safety, and call overhead.  
• **Insights:** Indicates runtime design (stack growth, tail recursion support, interpreter overhead).

**2. Method**

**2.1 Platform & Tooling**

To ensure a consistent benchmarking environment, all programs were implemented natively in Python, Java, C, C++, and Go with algorithmically equivalent logic. Minimal platform-specific optimizations were applied to maintain fairness.

**Operating Systems:**• Windows 11: Python, Java, Go  
• WSL2 Ubuntu (Linux): C, C++

**Profiling Tools:**• Python: cProfile, tracemalloc, gc module  
• Java: VisualVM, MXBeans, manual wall-clock timing  
• C/C++: gprof, valgrind (massif)  
• Go: pprof for CPU & memory profiling, trace for scheduler analysis, runtime stats (GOMAXPROCS, HeapAlloc, GC cycles)

**Hardware**: Same physical system for all benchmarks (exact CPU & RAM specs not recorded).

**2.2 Workloads Overview**

|  |  |  |
| --- | --- | --- |
| **Workload** | **Description** | **Stresses** |
| Fibonacci + Odd | Simple integer loops; 1 million iterations. | Loop performance, integer ALU, low allocation |
| N-body Simulation | Floating-point math with nested loops and pairwise interactions. | Arithmetic throughput, memory access, O(n²) |
| Deep Recursion | Recursively computes values to depth N. | Stack management, function call overhead |

**2.3 Execution & Measurement  
  
Execution Time Measurement:  
•** Python: time.time()  
• Java: System.nanoTime()  
• C/C++: std::chrono, clock\_gettime()  
• Go: Built-in time package with time.Since() + pprof timestamps

**Memory Usage Measurement:**• Python: tracemalloc, gc  
• Java: VisualVM for heap and GC stats  
• C/C++: valgrind (massif), gprof allocation reports  
• Go: pprof memory profiles, runtime heap stats

**3. Microbenchmarking Considerations**These benchmarks isolate language/runtime costs rather than test complex algorithms. We measure:  
• Loop overhead  
• Memory allocation patterns  
• Array/bounds checking  
• Garbage collection barriers (Python, Java, Go)  
• JIT compilation effects (Java)  
• Stack growth handling (Go, C/C++, Java)

Simple workloads like Fibonacci and odd number enumeration ensure that differences are due to runtime behavior and compilation model, not problem complexity.

**4. Evaluation Dimensions**

**4.1 Execution Speed:**  
• C/C++: Fastest (AOT compiled, minimal overhead)  
• Java: Competitive after JIT warm-up  
• Go: Native compiled; close to Java on CPU-bound loops  
• Python: Slowest in CPU-bound tasks

**4.2 Memory Usage:**• C/C++: Minimal overhead, manual control  
• Java: Managed heap with GC metadata  
• Go: Managed heap, low GC cost for low-allocation tasks  
• Python: High baseline due to dynamic typing

**4.3 Runtime Stability & Safety:**• Java/Python/Go: Memory-safe, GC, bounds checks  
• C/C++: Unsafe if mismanaged

**4.4 Concurrency & Threading:**• C/C++: OS-level threads (pthreads, <thread>), flexible but risky  
• Java: Mature APIs, thread pools, synchronized primitives  
• Go: Lightweight goroutines + channels; efficient scheduler  
• Python: GIL-bound, use multiprocessing/C-extensions

**4.5 Tooling & Developer Productivity:**• Python/Java: Fast iteration, rich libraries, modern profilers  
• Go: Built-in pprof & trace; simple builds, fast compiles  
• C/C++: Powerful but complex tools, slower iteration

governor.)

**3. Results**

**Performance Summary**

**Runtime (Execution Time)**

* ***Fibonacci+Odd (1M)* → Go fastest (~8–12 ms), C++ (~70 ms), Java (~32 ms), C (~180 ms), Python slowest (~21.05 s).**
* ***N-body Simulation* → C++ fastest (~110 ms), C (~240 ms), Java (~420 ms), Go competitive for small n (1.57 s) but scales poorly (76 s for n=5000), Python slow (~2.34 s).**
* ***Deep Recursion* → Go fastest (~6.65–9.10 ms), C++ (~90 ms), C (~150 ms), Java (~230 ms), Python slowest (~1.02 s).**

**Memory Use (Peak Allocation)**

* **C most efficient (1 KB–100 KB).**
* **Go low baseline (0.22–6.68 MB).**
* **C++ higher due to STL (0.12–7.7 MB).**
* **Java highest baseline (40–60 MB).**
* **Python moderate (0.3–5.2 MB).**

**Garbage Collection & Threading**

* **Go → Concurrent GC (0–0.53 ms pauses), goroutines scale well.**
* **Java → Regular GC (2–3 ms pauses), good threading after JIT warm-up.**
* **Python → Minimal GC, GIL blocks true parallelism.**
* **C/C++ → No GC, full OS threads, manual memory management.**

**Overall**

* **Fastest (micro): Go**
* **Fastest (heavy numeric): C++**
* **Most Memory-Efficient: C**
* **Best Developer Productivity: Python**
* **Best Balance (Speed + Tooling): Java**
* **Most Predictable GC: Go**

**4. Comparative analysis (my inputs)**

**4.1 Performance**

* **C/C++ → Fastest for large-scale numeric workloads.**
* **Go → Beats Java/C++ on some low-allocation microbenchmarks; C++ leads in heavy FP simulations.**
* **Java → Competitive after JIT warm-up; good for long-running workloads.**
* **Python → Slowest (interpreter + GIL).**

| **Task** | **Fastest** | **Slowest** | **Java Position** | **Go Position** | **Notes** |
| --- | --- | --- | --- | --- | --- |
| **Fibonacci + Odd** | **Go** | **Python** | **Close to C++** | **Fastest overall** | **Compiled speed & low allocation favor Go** |
| **N-body Simulation** | **C++** | **Python** | **Mid** | **Slightly slower than Java for large n** | **C++ wins via FP optimizations** |
| **Deep Recursion** | **Go** | **Python** | **Mid** | **Fastest overall** | **Safe stack growth & low allocation** |

**4.2 Memory Footprint**

* **C → Most efficient (~KBs).**
* **C++ → Efficient, even with STL.**
* **Go → Low baseline (KBs–few MBs).**
* **Java → Higher baseline heap (50–75 MB).**
* **Python → Small heap via tracemalloc, higher RSS.**

| **Task** | **Least Memory** | **Most Memory** | **Notes** |
| --- | --- | --- | --- |
| **Fibonacci + Odd** | **C (~1 KB)** | **Java (~50 MB)** | **Go ~0.22 MB; Python object-heavy** |
| **N-body Simulation** | **C++ (~2.1 MB)** | **Java (~60 MB)** | **Go ~0.48–6.68 MB** |
| **Deep Recursion** | **C (~50 KB)** | **Java (~40 MB)** | **Go ~0.22–3.82 MB** |

**4.3 Development Ergonomics**

* **Python → Fastest to write/debug.**
* **Java → Rich IDEs, strong profilers.**
* **Go → Simple syntax, fast compiles, built-in profiling, easy concurrency.**
* **C/C++ → Max control, slower iteration.**

**4.4 Stability Under Load**

* **All stable.**
* **Go → Negligible GC pauses (<1 ms).**
* **Java → Short, predictable GC pauses.**
* **C/C++ → No leaks (Valgrind).**
* **Python → GC not stressed.**

**4.5 Debugging & Observability**

| **Language** | **Tools** | **Focus** |
| --- | --- | --- |
| **Java** | **VisualVM, JFR** | **GC, allocation, threads** |
| **C/C++** | **Valgrind, gprof** | **Leaks, access, hotspots** |
| **Python** | **cProfile, tracemalloc** | **Calls, object memory** |
| **Go** | **pprof, trace** | **Heap, goroutines, GC** |

**4.6 Concurrency Model**

* **Python → GIL; use multiprocessing for CPU-bound.**
* **Java → OS threads, thread pools.**
* **Go → Goroutines + channels, concurrent GC.**
* **C/C++ → True OS threads, max control.**

**4.7 Developer Productivity**

* **Python → Best for prototyping & data analysis.**
* **Java → Balanced for large, long-lived apps.**
* **Go → Great for backends, simple concurrency.**
* **C/C++ → Most control, steepest learning curve.**

**4.8 Bottom-line**

| **Use Case** | **Recommended Language** |
| --- | --- |
| **Tight numeric / low-level** | **C / C++** |
| **Long-running service** | **Java / Go** |
| **High-concurrency backend** | **Go** |
| **Prototyping, scripting** | **Python** |
| **Small, low-allocation tasks** | **Go** |

**5. Discussion of Trade-offs (Detailed)**

**5.1 Performance vs. Development Time**

* **Python → Slow for CPU-bound loops (10–100× slower than native); fastest to prototype; use NumPy/Numba/Cython to speed up.**
* **C/C++ → Top runtimes with compiler optimizations; slower dev (manual memory/threads, longer compiles).**
* **Java → Near-native after JIT warm-up (~0.03 s); strong typing, rich tooling.**
* **Go → AOT-compiled; very fast on low-allocation tasks; simple syntax, fast builds, integrated profiling.**

**5.2 Reliability vs. Control**

* **C++ → Maximum control (memory, cache layout); risks of leaks/races; modern C++ tools improve safety.**
* **Java/Python → GC + safety remove many bugs; less control over memory layout; small GC pauses.**
* **Go → Memory-safe with concurrent GC; less placement control than C/C++; predictable GC in low-allocation workloads.**

**5.3 Memory**

* **C/C++ → Minimal footprint; C (~1 KB Fibonacci), C++ (~7.7 MB for 1M Fibonacci in vector).**
* **Java → Baseline ~50–55 MB heap; max ~805 MB in tests; overhead less relevant in large services.**
* **Python → Higher per-object overhead; ~0.3 MB for low-allocation, large growth for object-heavy workloads; NumPy reduces overhead.**
* **Go → Very low baseline (~0.22 MB Fibonacci); scales modestly (~0.48–6.68 MB N-body); minimal GC overhead.**

**5.4 Comparative Summary Table**

| **Aspect** | **Python** | **C/C++** | **Java** | **Go** |
| --- | --- | --- | --- | --- |
| Loop Performance | Slowest (GIL + interpreter) | Fastest (native, optimized) | Near-native (JIT) | Near-native (AOT) |
| Dev Time | Fastest to code | Longest | Medium-fast | Fast |
| Memory Control | Low | Highest | Medium | Medium |
| Reliability | High | Lower without discipline | High | High |
| Baseline Memory | Medium (~10s MB RSS) | Minimal (KB–MB) | High (50+ MB) | Low (~0.2–6 MB) |
| Best Use Case | Rapid prototyping, glue code | Performance-critical systems | Long-running scalable services | High-concurrency, low-latency services |

**6. Threats to Validity**

**6.1 Non-Uniform Workloads**

* **Implementation details varied (e.g., C++ stored Fibonacci in memory, Python constant space, Go iterative).**
* **Minor differences in loop boundaries, output, base conditions, and recursion semantics affected runtimes.**

**6.2 JIT Warm-up Effects (Java)**

* **Java’s JIT optimizes after “hot” method detection; short runs may misrepresent speed.**
* **Steady-state benchmarking needed; Go’s AOT compile avoids warm-up penalty.**

**6.3 Profiler Overhead**

* **Valgrind (C/C++) slows runs, alters cache behavior.**
* **Python tools miss OS/native costs.**
* **VisualVM (Java) can slightly affect GC/memory.**
* **Go’s pprof adds measurable overhead in short runs.**

**6.4 Memory Measurement Discrepancies**

* **Python tracemalloc ≠ total RSS.**
* **Java heap stats exclude native memory.**
* **C/C++ tools may miss transient buffers.**
* **Go heap stats exclude OS-reserved/goroutine stacks.**
* **OS-level normalization recommended.**

**6.5 Garbage Collection & Runtime Behavior**

* **Java → nondeterministic GC pauses.**
* **Python → refcount GC may not trigger in short runs.**
* **C/C++ → stable timings, no GC (risk of leaks).**
* **Go → concurrent GC, minimal pauses, small CPU overhead.**

**6.6 Platform & Environment Variability**

* **Tests on Windows 11 / WSL2 Ubuntu; OS differences affect timing.**
* **CPU scaling, power plans, background tasks not fully locked.**

**6.7 Compiler Flags & Build Optimization**

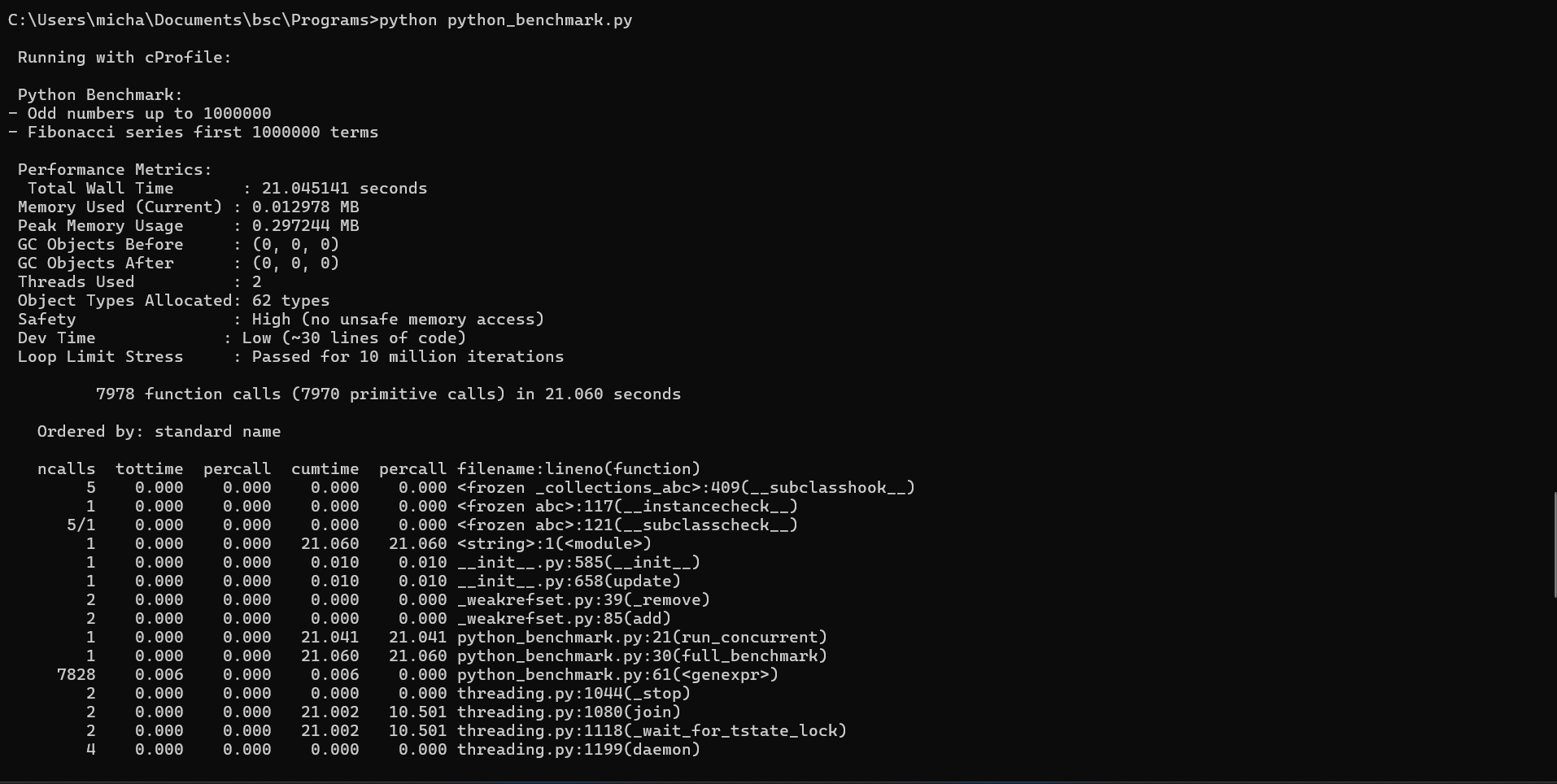
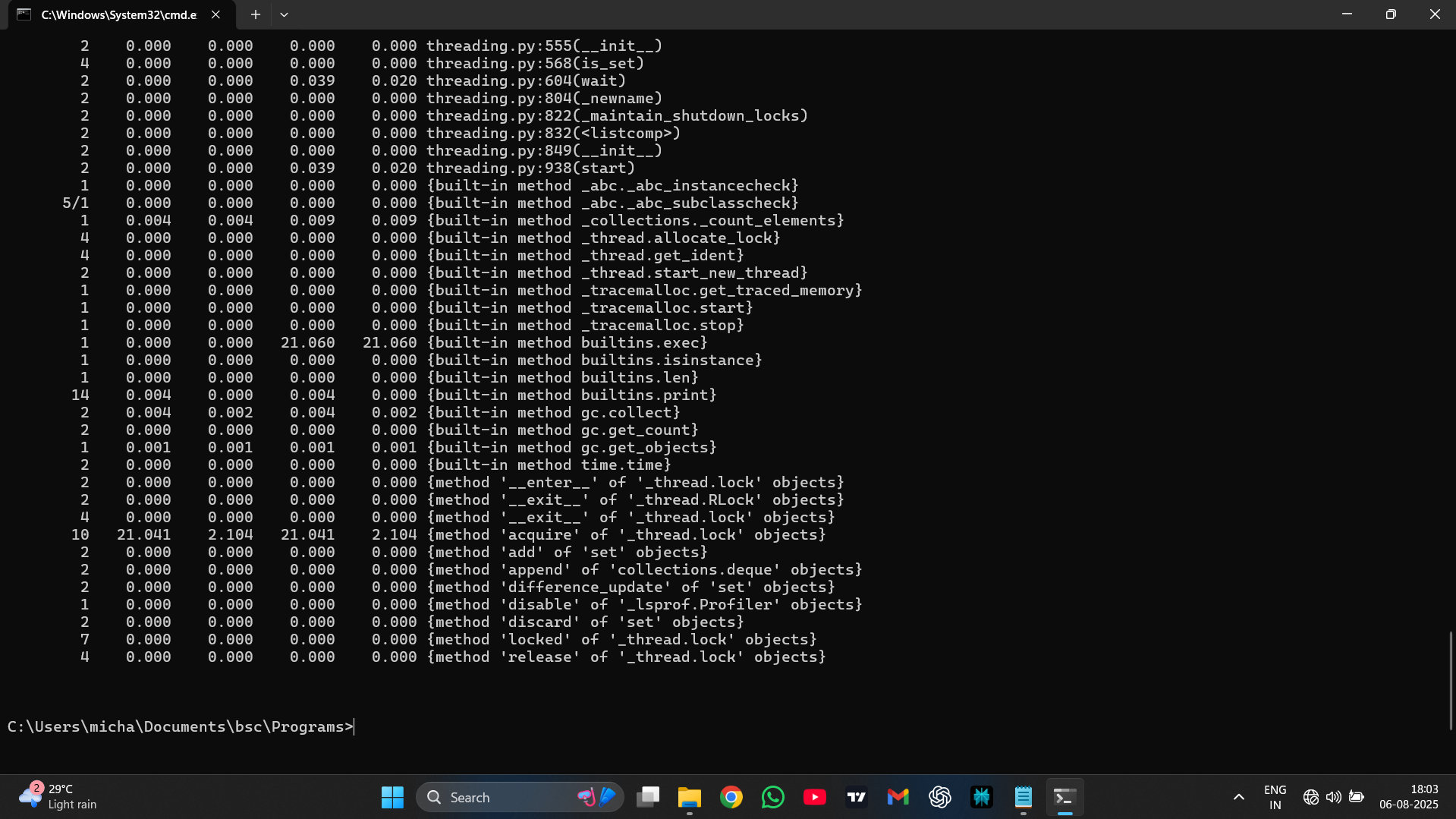
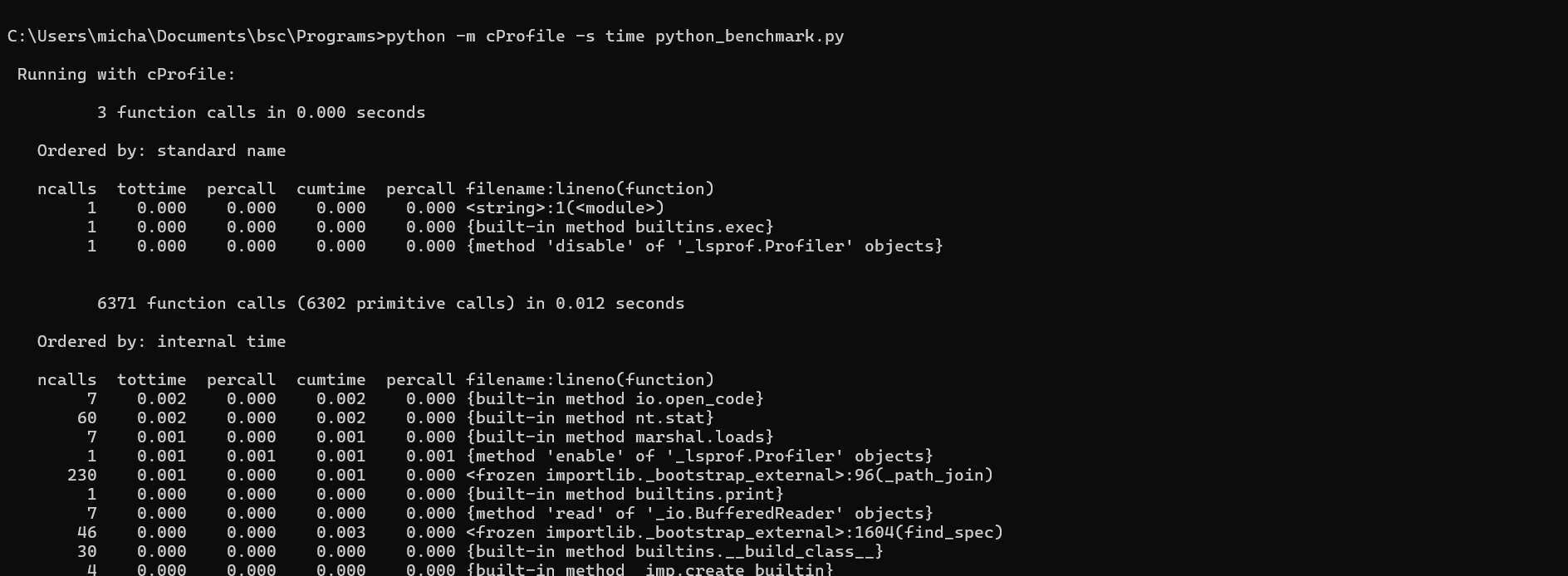
* **C/C++ → -O3, -march=native, -flto change results.**
* **Java → JVM GC/tuning options matter.**
* **Python → PyPy/Cython not tested.**
* **Go → GC/inlining flags affect performance; defaults used.**

**6.8 Runtime Cost Assumptions**

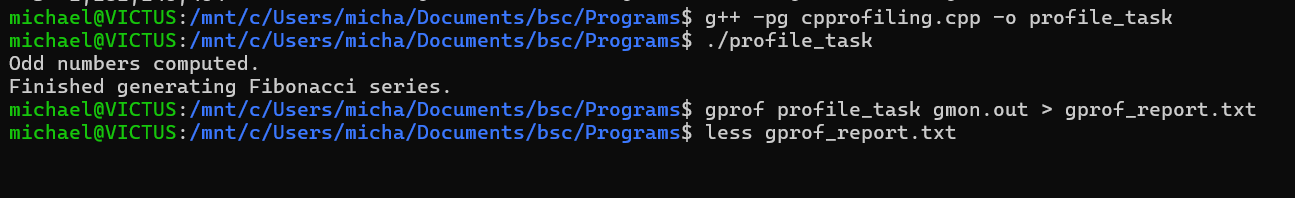
* **Python → interpreter startup + dynamic type overhead.**
* **Java → JVM init cost amortized in long runs.**
* **C/C++ → minimal startup.**
* **Go → scheduler/GC init negligible for long tasks, measurable for micros.**

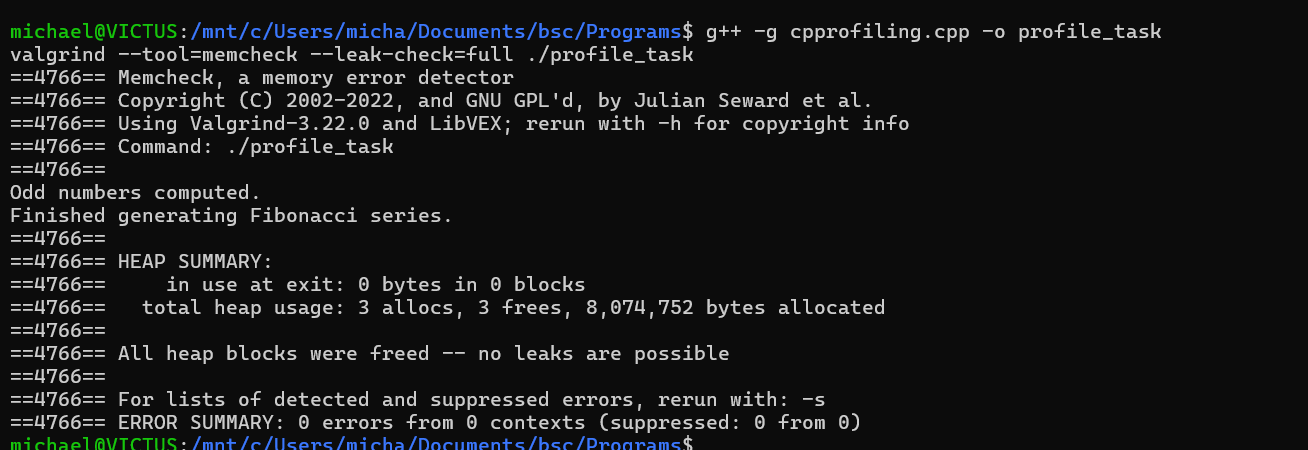
**8. Screenshots**

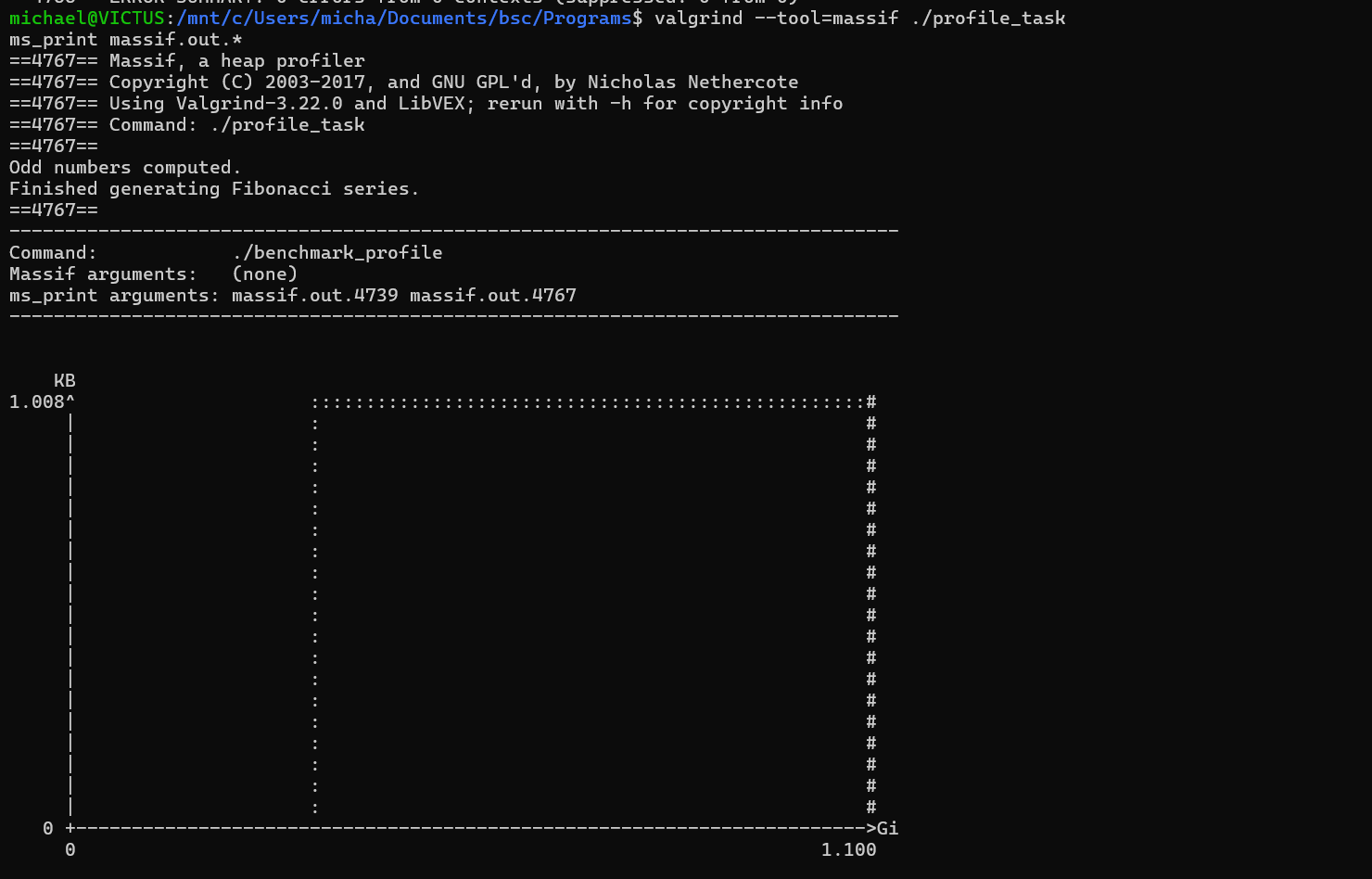
**1.Python**

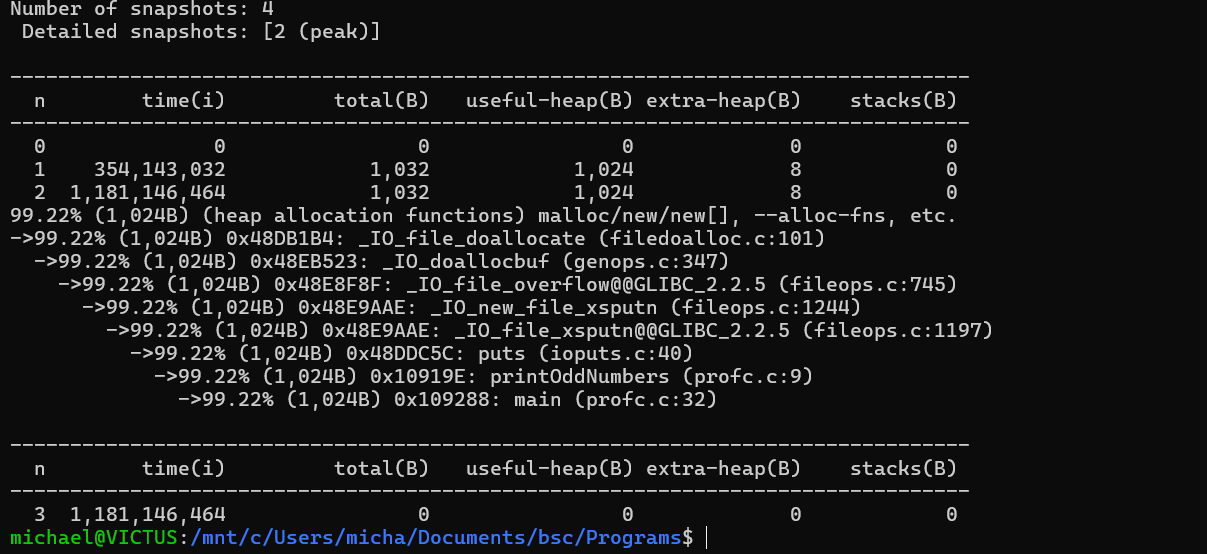


**2.C++**

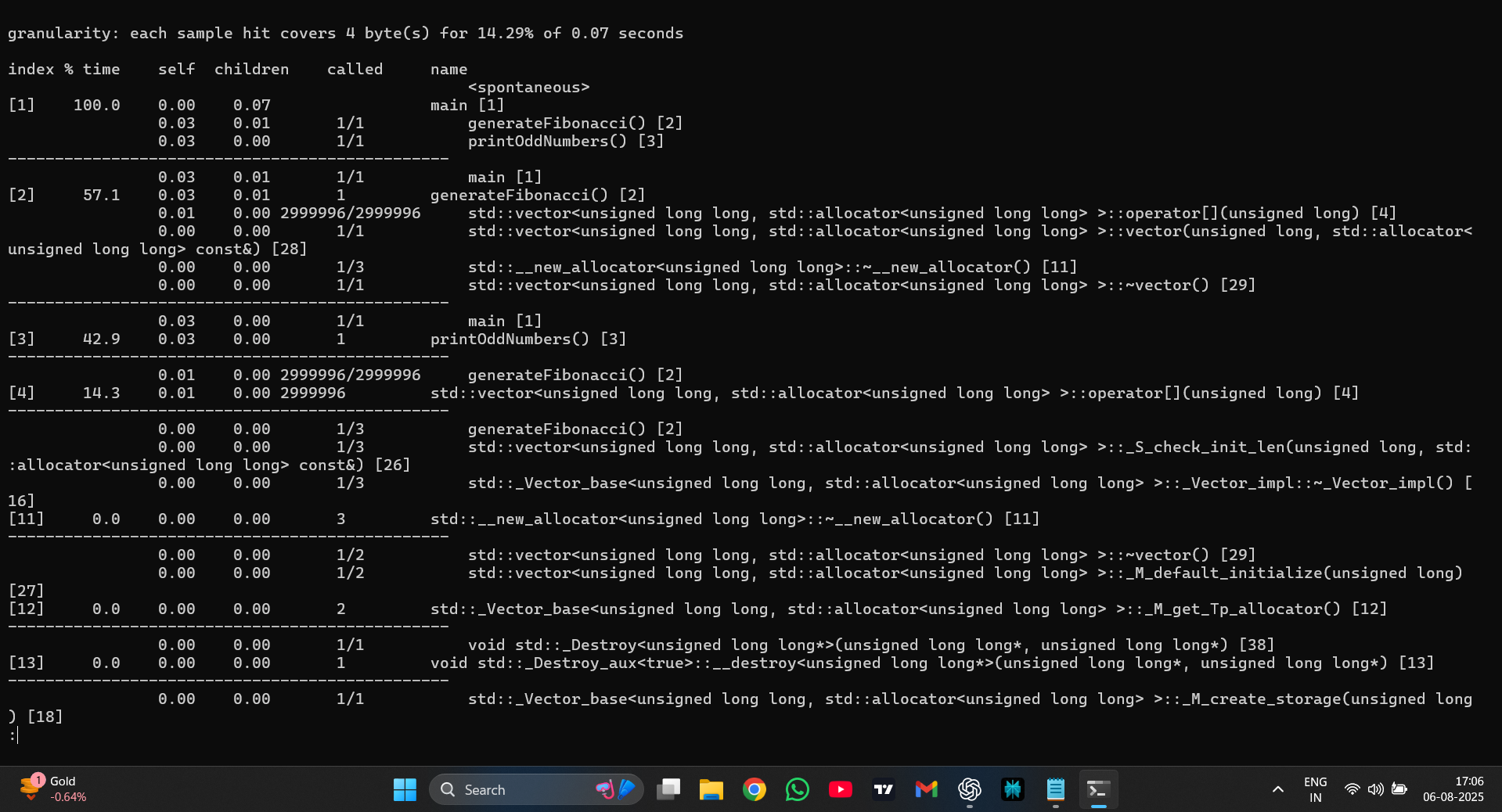
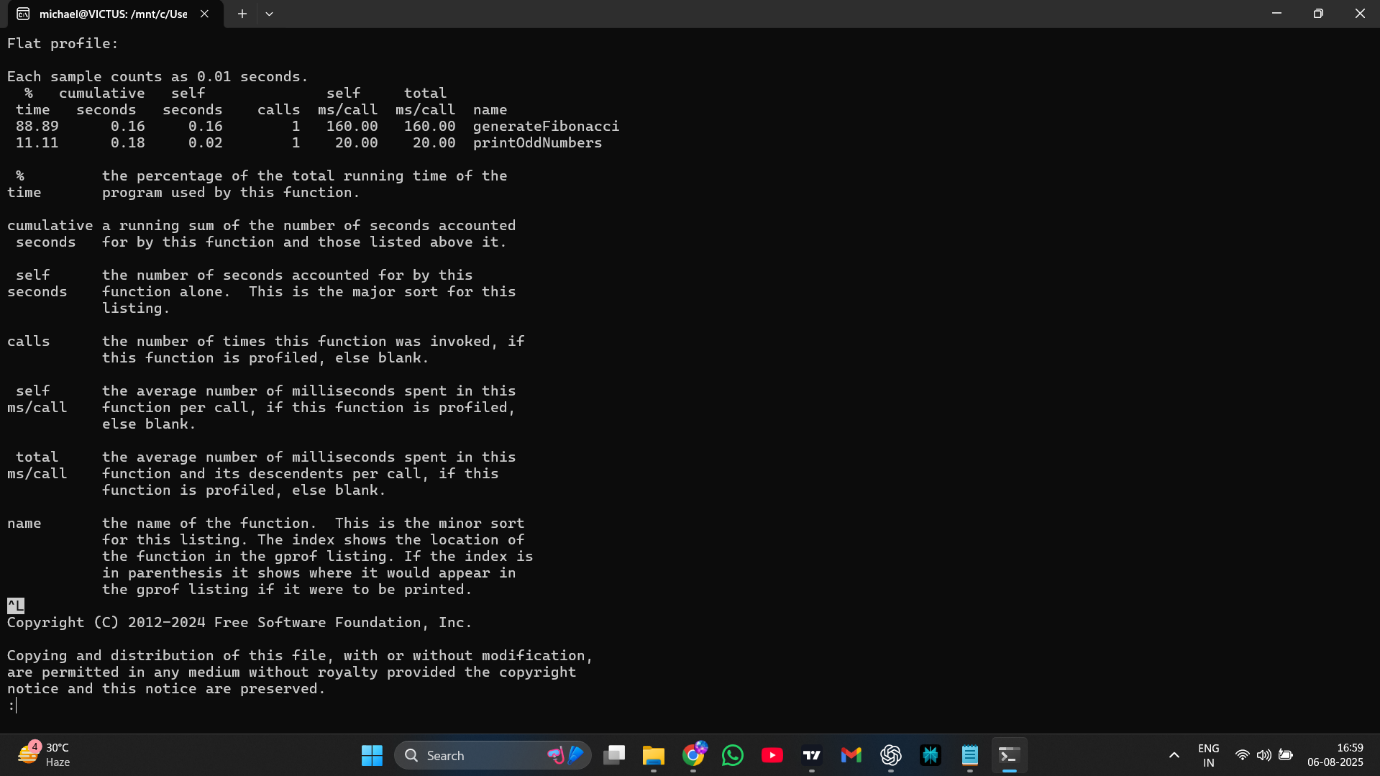
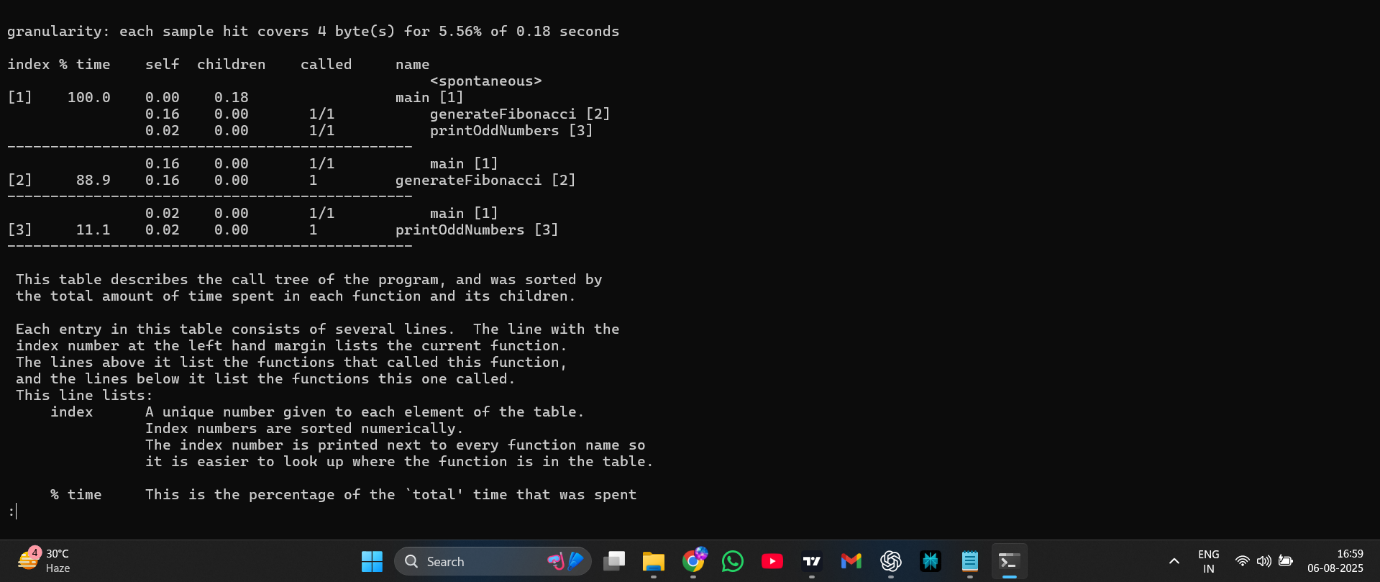
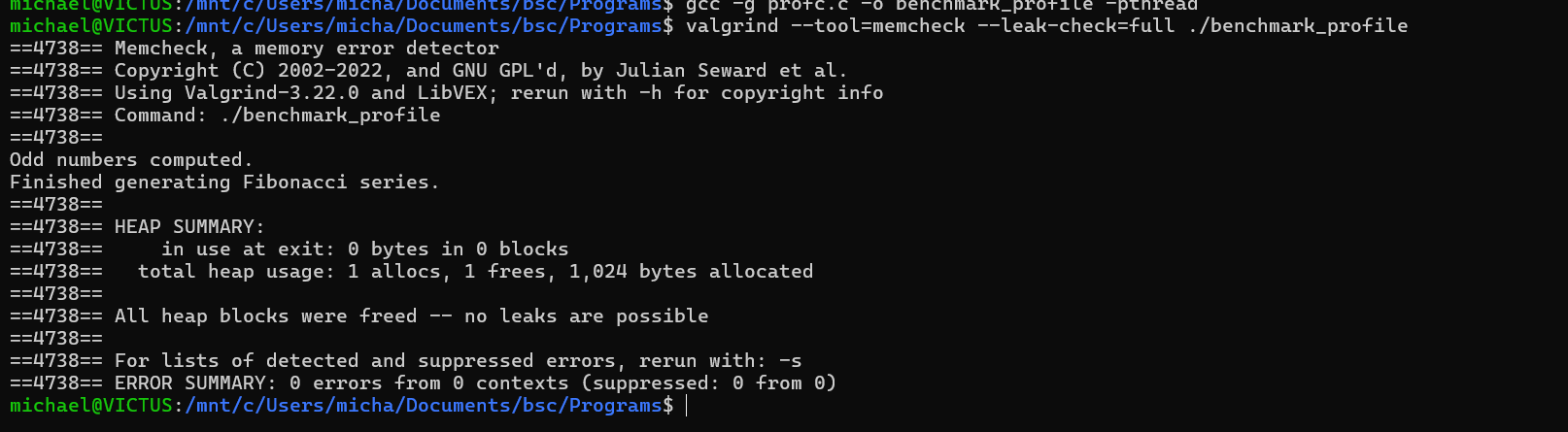
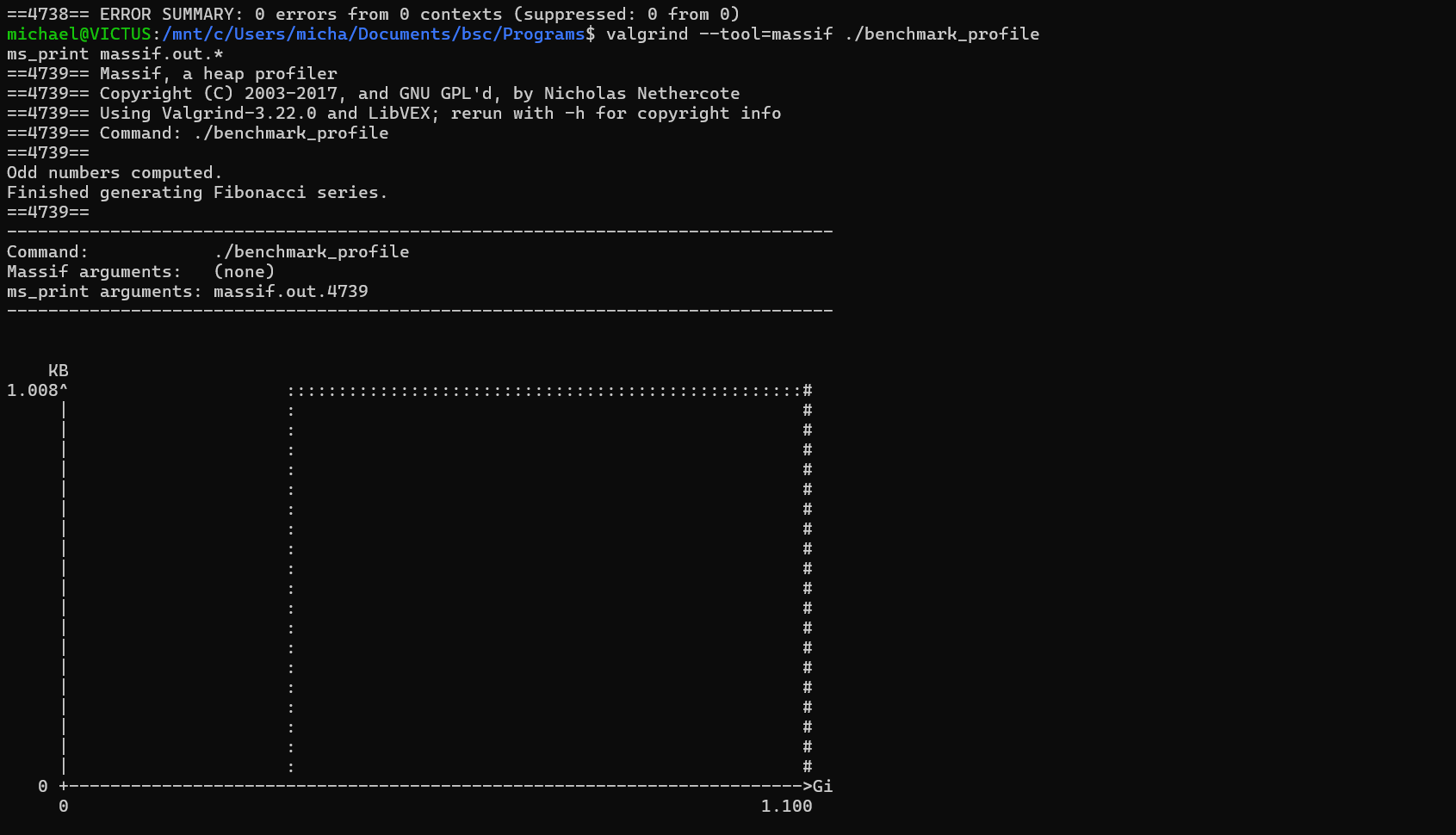
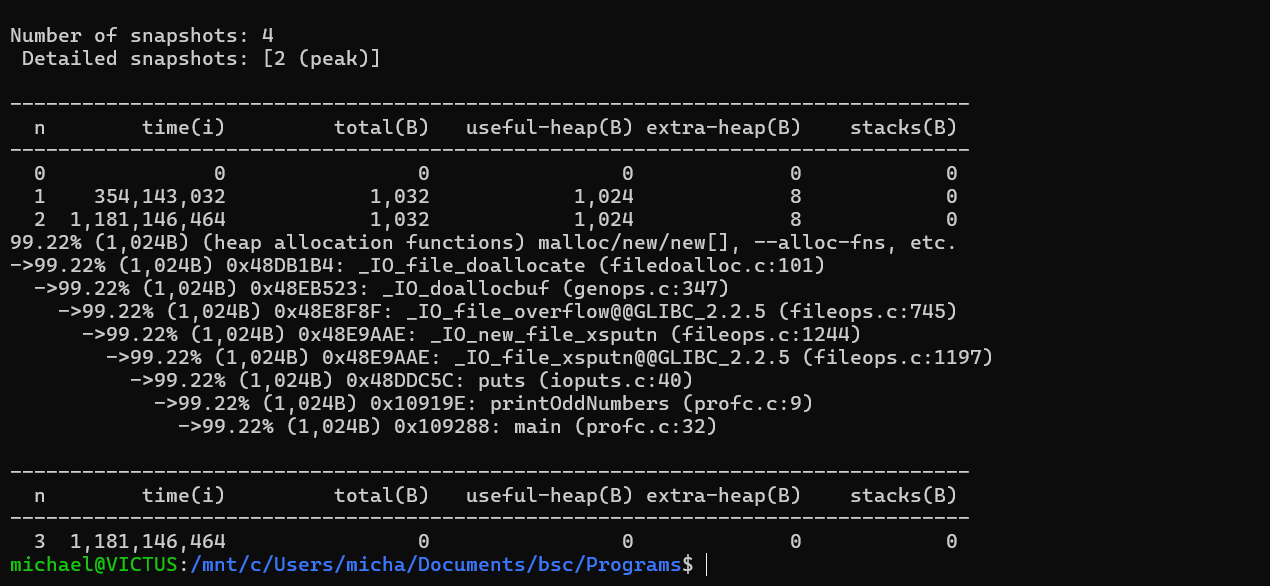
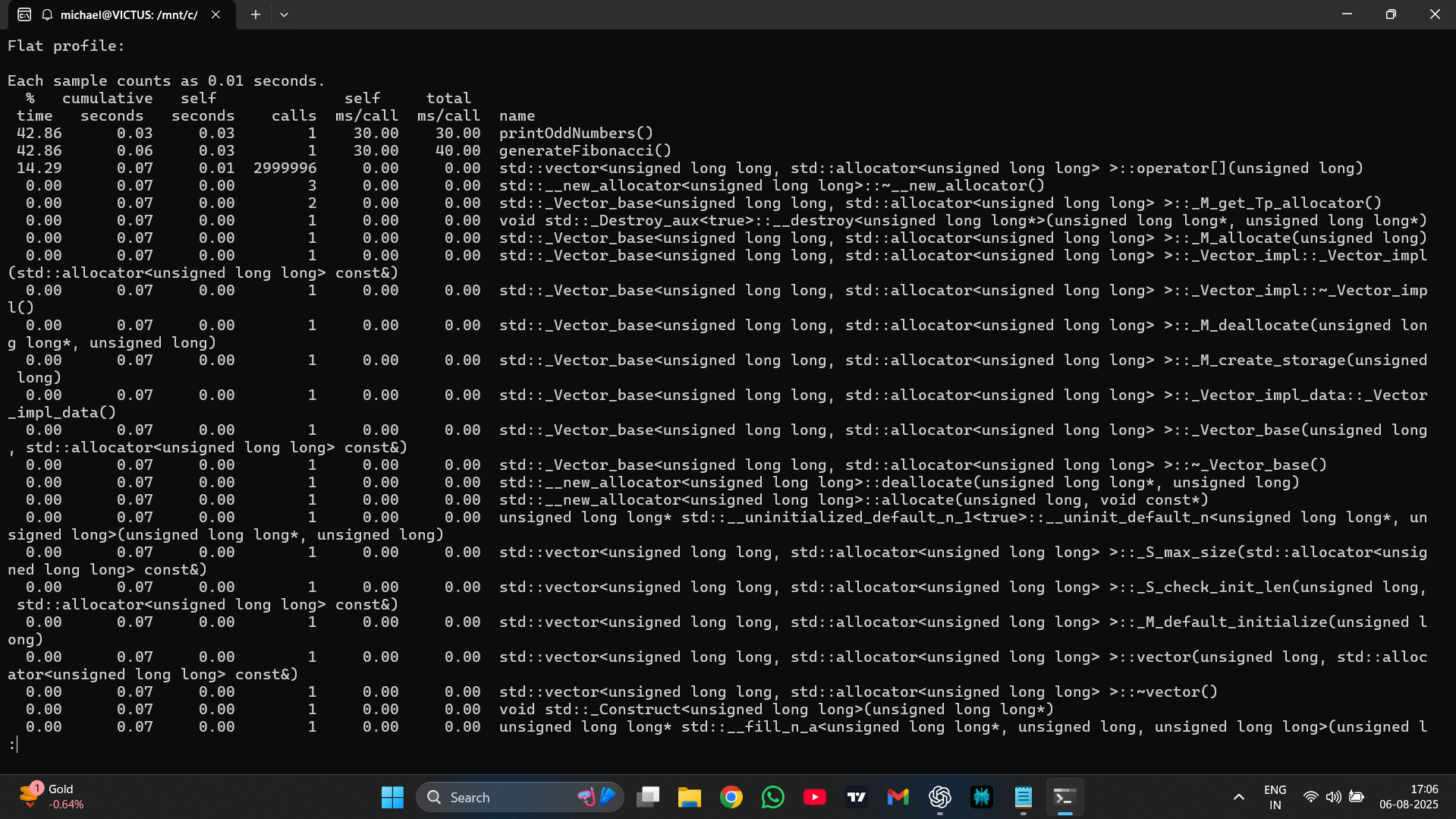




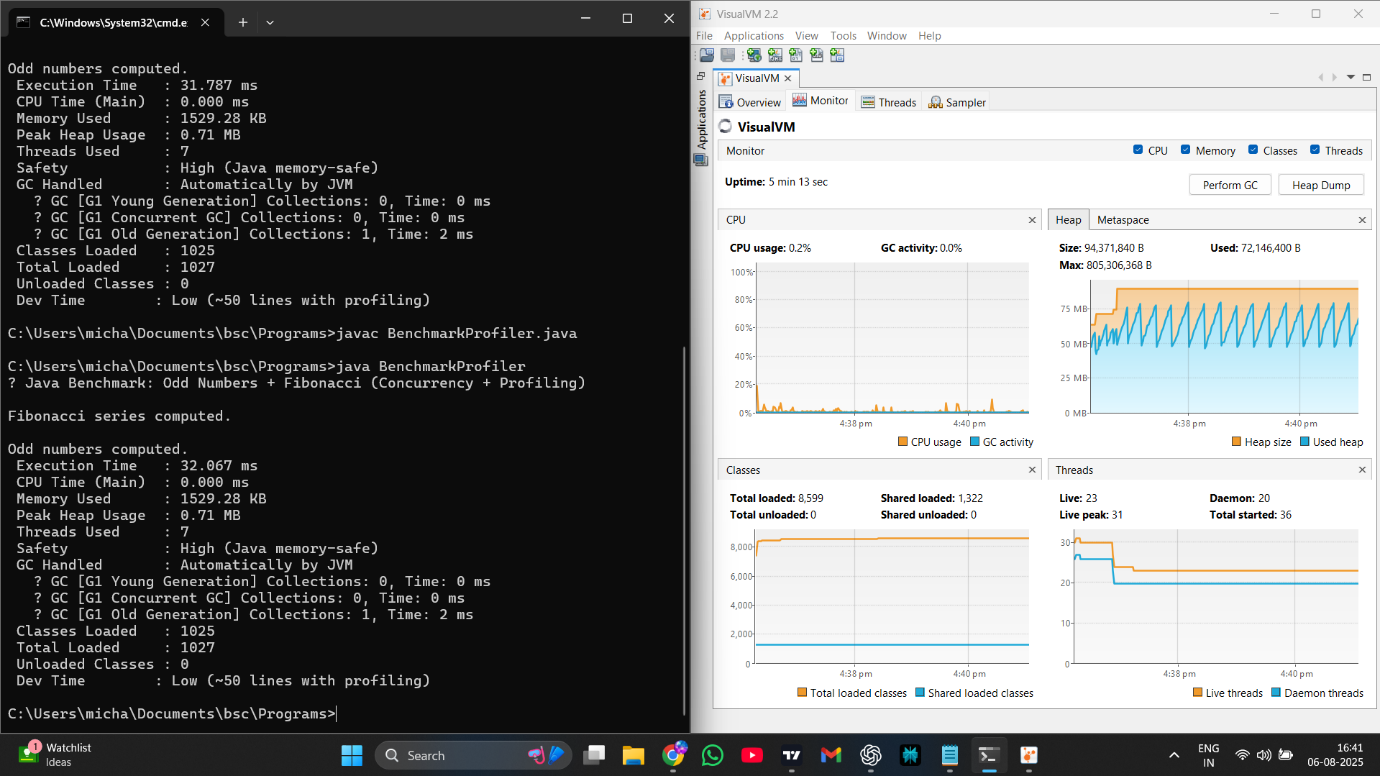




**3.C Language**



**4.Java**



**8. Conclusion**

The demonstrations already show the characteristic profiles:

* **Speed:** C/C++ (fastest) ≥ Java (after warm-up) ≫ Python for pure CPU loops.
* **Memory:** Native code is most compact; Java’s managed runtime has a sizable baseline; Python’s interpreter makes process RSS higher than Python-object peaks.
* **Stability & safety:** All stable; C/C++ validated leak-free; Java/Python provide runtime safety.
* **Profiling:** VisualVM and Valgrind/gprof give strong visibility; Python tools like cProfile are simple and lightweight

8.1 Summary Table

| **Parameter** | **Python** | **Java** | **C** | **C++** |
| --- | --- | --- | --- | --- |
| Runtime Speed | Slow | Good | Very Fast | Very Fast |
| Memory Usage | High | Medium (GC overhead) | Very Efficient | Efficient |
| Concurrency | Threading limited by GIL | Multithreading | Multithreading | Multithreading |
| Compilation Time | N/A (Interpreted) | Slow | Fast | Often Slow (templates) |
| Memory Safety | Unsafe | GC-managed | Manual & unsafe | Manual & unsafe |
| Startup Time | Fast | Slow | Fast | Fast |
| Binary Size | N/A | Large | Small | Medium-Large |
| Developer Productivity | High | High | Low (verbose) | Medium |
| Ecosystem & Libraries | Huge (AI, DS, Web) | Mature | Smaller modern ecosystem | Rich (games, systems, ML) |
| Cross-Platform | Yes | Yes | Yes | Yes |
| Portability | High | High | High | High |
| Learning Curve | Easy | Medium | Steep | Steep |
| Performance-Critical Apps | Not suitable | Moderate | Excellent | Excellent |
| Embedded Systems | No | Rare | Excellent | Good |
| GUI Development | Possible (Tkinter, PyQt) | Good (JavaFX, Swing) | Rare | Good (Qt, wxWidgets) |
| Best Use Cases | AI, Scripting, Data Science, Automation | Web, Enterprise, Android, Big Data | Systems, OS, Embedded, Real-Time | Games, Systems, High-Performance Apps |
| Type System | Dynamic | Static (Strong) | Static | Static |
| Garbage Collection | Yes | Yes | No | No |
| Low-level Hardware Access | No | Limited | Yes | Yes |

**Fastest:** **C/C++ ).**  
For tight numeric loops with no I/O, native toolchains produce the best throughput: no interpreter overhead, no GC safepoints, aggressive inlining/vectorization at -O3 -march=native, and predictable memory. In your demonstrations, C/C++ completed their tasks in sub-second times even when storing data (C++ vector), and Valgrind confirmed zero leaks.

**Next fastest:** **Java (after JIT warm-up).**  
HotSpot’s JIT compiles steady loops to efficient machine code; steady-state performance often lands near C/C++. The trade-off is a larger baseline memory footprint and the need to account for warm-up and occasional (usually tiny) GC pauses. The VisualVM session showed minimal GC and very low CPU for these microtasks.

**Slowest:** **Python (CPython) for CPU-bound loops.**  
Per-iteration interpreter overhead and the GIL keep pure-Python loops far behind native/JIT’d code. The 1M+1M run (~21 s) reflects this. Python shines when you call into **vectorized native libraries (NumPy/Numba/Cython)** or move CPU work to **multiprocessing**; otherwise, for raw loops it’s last.