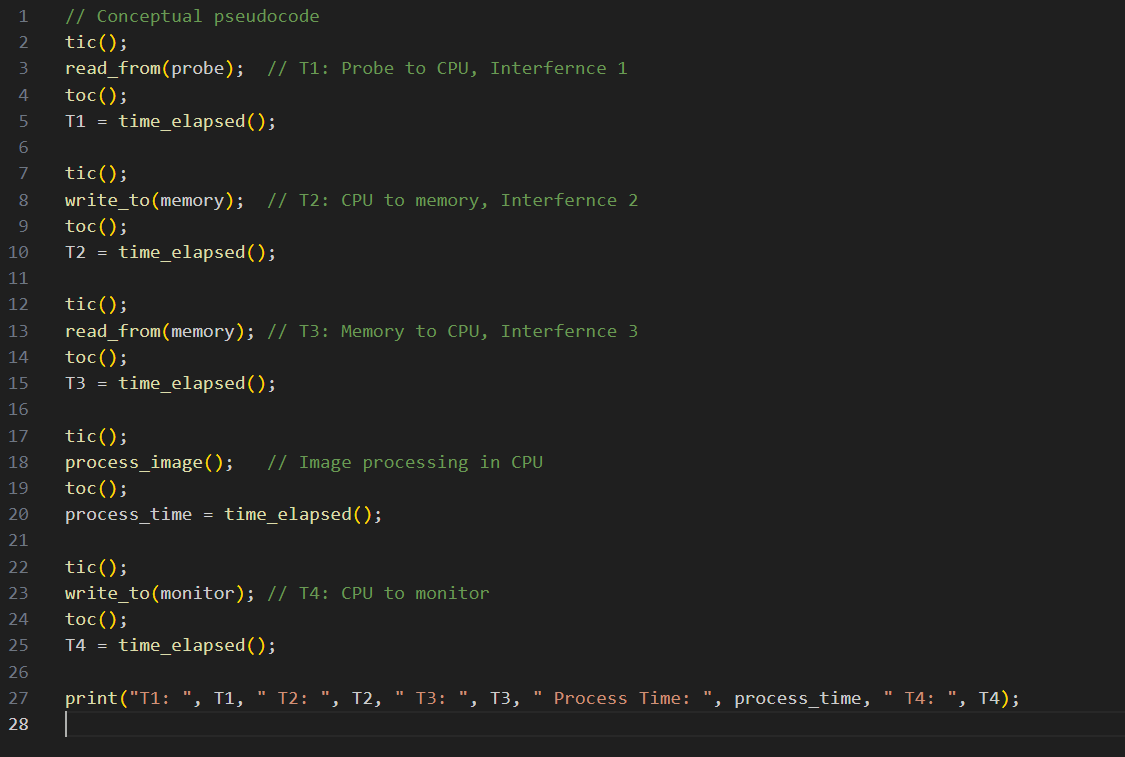
1)



2)

* Design level
* Algorithm and data structure level
* Source code level
* Build level
* Compile level
* Assembly level
* Run time (e.g., Just-in-Time compilation)

**General order:** Starting from higher levels (design, algorithms) to lower levels (source, build, compile, assembly).

Since refinement at higher levels often has a larger impact and is harder to change later and could result in a complete re-writing.

3)

* CPU**-bound:** Task that depend heavily on the CPU, with long CPU bursts and infrequent I/O (or any other) operations. Performance improves by increasing CPU speed.
* **I/O-bound:** Tasks depend on the input/output system, with frequent, short CPU work with long waits for I/O between each CPU brust. Performance improves by speeding up I/O operations.

**4)**

* Resolution: X×YX \times YX×Y pixels per image, 1 byte per pixel
* Frame rate: 15≤FPS≤30
* Latency: No more than 500 ms
* Size/weight: Comparable to a regular PC

5)

Optimizations often add complexity (e.g., special cases, tricks) and reduce readability, making debugging and updates more difficult.

Still, we will accept this trade-off, when achieving critical performance benchmarks or resolving bottlenecks in time-sensitive systems.

**How to minimize the negative impact:**

* Use modular and well-documented code.
* Isolate optimized code sections and add comments/refers to explain them.

6)

To identify a bottleneck in a complex system we can either use profiling tools to measure resource utilization or analyse critical sections of the code (hot spots) and identify resource constraints (e.g., CPU, memory, I/O).

To the primary performance limiter after identifying the bottleneck

* Experiment with changes to the suspected bottleneck and measure the performance impact.
* Validate those improvements in the bottleneck lead to overall system performance gains.
* Rule out other potential bottlenecks through systematic testing and analysis.