Amazon Interviewer:
Hi Can, welcome back. Let's do a DSA round in an Amazon operations context. I'll present the problem, then I'd like you to summarize it, ask clarifying questions, and think aloud as you solve it.

Problem — Minimum Average Processing Time at a Sort Center

At an Amazon sort center, each customer order turns into a **job** when it arrives. We have **one packing station** (single machine), jobs are **non-preemptive** (once started, you must finish), and you may pick **any available job** when the station is free.

For each job `i`:

- `arrival[i]` — time it enters the system,
- `duration[i]` — uninterrupted processing time the station needs.

Goal: choose a processing order that **minimizes the average turnaround time** (also called average completion delay), defined for job `i` as:

```
turnaround(i) = completion_time(i) - arrival[i]
```

We return the **floor** of the average over all jobs.

If the station is idle and **no job has arrived yet**, it **waits** until the next arrival.

Return: an integer = `floor( average(turnaround(i)) )`.

Constraints

- `1 ≤ n ≤ 2 * 10^5`
- `0 ≤ arrival[i], duration[i] ≤ 10^9`
- Single machine; non-preemptive

Worked Examples (with explanations)

Example 1

`arrival = [0, 1, 2]`

`duration = [3, 9, 6]`

Optimal schedule (Shortest-Job-First among available):

- `t=0`: only job0 (dur 3) is available → run job0, completes at `t=3`, turnaround = `3 - 0 = 3`.
- `t=3`: jobs1(dur 9, arr 1), job2(dur 6, arr 2) are available → pick shorter job2 → completes at `t=9`, turnaround = `9 - 2 = 7`.
- `t=9`: run job1 → completes at `t=18`, turnaround = `18 - 1 = 17`.

Sum = `3 + 7 + 17 = 27`, average = `27 / 3 = 9`, **answer = 9**.

Example 2

`arrival = [0, 0, 0]`

`duration = [2, 2, 2]`

All three jobs arrive at time 0 and take 2 time units each. Any order yields:

- First completes at `t=2` → turnaround `2 - 0 = 2`
- Second completes at `t=4` → `4 - 0 = 4`
- Third completes at `t=6` → `6 - 0 = 6`

  Sum = `2 + 4 + 6 = 12`, average = `12 / 3 = 4`, **answer = 4**.

Note: We're using **turnaround time** (completion − arrival). If you meant *queue waiting only*, we'd define it differently. For this round we stick to turnaround, as above.

Candidate (Excellent) — Problem Summary:
We have `n` non-preemptive jobs on a single machine. Each job has an arrival and a processing duration. When the machine is free, it can pick **any** job whose arrival time has passed. We want to minimize the **average turnaround** = completion − arrival, and return its floor. If no jobs are available, time advances to the next arrival. Constraints push us to an `O(n log n)`-type solution.

Did I capture that correctly?

Interviewer:
Perfect.

Candidate — Clarifying Questions:

1. If multiple jobs are available at the same time, I'm free to choose any ordering?
2. Ties on duration don't matter for correctness, right?
3. Confirm we return `floor(total_turnaround / n)` (integer division)?

Interviewer:

1. Yes, any available job.
2. Correct; any tie-break among equal durations is fine.
3. Yes, floor of the average.

Candidate — Approach Exploration

Brute-force (reject): Trying all valid schedules is factorial and intractable.

Greedy insight: To minimize average turnaround (equivalently, sum of completion minus arrival), when jobs are available, always process the **shortest duration next**. This is the classic **Shortest Job First (SJF)** strategy.

Pattern (used because it fits and optimizes):

- **Greedy Algorithms (#27)** — locally optimal choice (shortest available job) leads to global optimum.

- Backed by an **exchange argument**: if you ever run a longer job before a shorter one that's already available, swapping them does not increase (and typically reduces) the total completion delay.

Data structure: A **min-heap (priority queue)** keyed by `duration` to select the next shortest job among those that have arrived.

Now I'll turn this into a **concrete step-by-step plan using Example 1** (`arrival=[0,1,2]`, `duration=[3,9,6]`):

1. **(We can skip a separate sort)** because `arrival` is already ascending.

   → *State now:* `arrival[i], duration[i]` represent job `i`, in arrival order.

2. **Maintain `time` and a pointer `i` over the arrays.**

   → Initialize: `time = 0`, `i = 0`, min-heap `pq = []`, `total_turnaround = 0`.

3. **Push all jobs with `arrival[i] ≤ time` into the min-heap `(duration[i], arrival[i])`.**

   ○ At start: `time=0`, `arrival[0]=0 ≤ time` → push `(3, 0)`.

   ○ Heap now: `[(3, 0)]`.

4. **If heap is empty, jump `time` to `arrival[i]`.**

   ○ *Explanation:* This models idling. If no arrived jobs are available, we fast-forward to the next arrival to avoid spinning.

5. **Pop the shortest job; update time and turnaround.**

   ○ Pop `(3, 0)` → run → `time = 0 + 3 = 3`

- Add turnaround: `time - arrival = 3 - 0 = 3` → `total = 3`.

6. **Repeat until all jobs processed.**
   - Now `time=3`, enqueue any arrivals `≤ 3`: jobs at `i=1 (arr=1,dur=9)` and `i=2 (arr=2,dur=6)` → push `(9,1)` and `(6,2)` → heap = `[(6,2),(9,1)]`
   - Pop shortest `(6,2)` → `time=9`, add `9-2=7` → total `= 10`
   - Pop `(9,1)` → `time=18`, add `18-1=17` → total `= 27`
7. **Return `total_turnaround // n`.**
   - `27 // 3 = 9` → matches Example 1.

This example-driven plan clarifies exactly what happens at each step.

This is `O(n log n)` due to sorting and heap operations.

Interviewer (Probe):
Give me the short correctness sketch for SJF here.

Candidate — Correctness (exchange argument, brief):

Consider two available jobs `A` and `B` with durations `a ≤ b`. If we schedule `B` then `A`, their contributions to total turnaround are larger or equal compared to `A` then `B`. Swapping any such inversions repeatedly yields a schedule where shorter jobs are always processed before longer ones among available jobs—i.e., SJF—without increasing the total turnaround. Hence SJF is optimal.

Candidate — Pseudocode (concise)

```
jobs = sort by arrival asc
i = 0
time = 0
total = 0
heap = min-heap of (duration, arrival)

while i < n or heap not empty:
    if heap empty and time < jobs[i].arrival:
        time = jobs[i].arrival
    while i < n and jobs[i].arrival <= time:
        push (jobs[i].duration, jobs[i].arrival) to heap
        i += 1
    d, a = pop heap
    time += d
    total += time - a

return total // n
```

Interviewer:

Please implement it and then we'll dry-run both examples.

Candidate — Code (Python)

```
import heapq
from typing import List


def min_average_turnaround(arrival: List[int], duration: List[int]) -> int:
    """
    Compute floor(average turnaround) on a single non-preemptive machine.

    Turnaround(i) = completion_time(i) - arrival[i]
    Strategy: Shortest-Job-First among jobs that have arrived.
              Use a min-heap keyed by duration to pick the next job.

    Assumptions for this interview:
      - arrival is already sorted ascending.
      - arrival and duration are aligned by index (job i is (arrival[i], duration[i])).
    """

    n = len(arrival)
```

```
if n == 0:
    return 0  # no jobs => average turnaround is 0 by convention

i = 0                    # pointer over (arrival, duration)
time = 0                 # current machine time
total = 0                # sum of all turnaround times
pq = []                  # min-heap of (duration, arrival) for available jobs

# Process until all jobs are scheduled (i == n) and heap is empty
while i < n or pq:
    # If no available jobs, jump time forward to the next arrival
    if not pq and i < n and time < arrival[i]:
        time = arrival[i]  # idle → fast-forward to next job arrival

    # Enqueue every job that has arrived by 'time'
    # Since arrival is sorted, this loop advances 'i' monotonically.
    while i < n and arrival[i] <= time:
        heapq.heappush(pq, (duration[i], arrival[i]))
        i += 1

    # If we have any available job, run the shortest duration next.
    if pq:
        d, a = heapq.heappop(pq)  # pick the job with smallest duration
        time += d                 # advance time by its duration
        total += (time - a)       # add its turnaround = completion - arrival
```

```python
    # Return the floored average turnaround
    return total // n
```

Interviewer:

Dry-run Example 1 and Example 2 exactly as stated earlier, and explain the outputs.

Candidate — Dry Runs

Example 1

`arrival = [0, 1, 2]`, `duration = [3, 9, 6]`

Sorted jobs: `(0,3), (1,9), (2,6)`

- `t=0`: push `(3,0)` → pop `(3,0)`, `t=3`, turnaround `3-0=3`, total=3
- `t=3`: push `(9,1)` and `(6,2)` → pop `(6,2)`, `t=9`, turnaround `9-2=7`, total=10
- `t=9`: pop `(9,1)`, `t=18`, turnaround `18-1=17`, total=27

  Average = `27/3 = 9` → **9**

Example 2

`arrival = [0, 0, 0]`, `duration = [2, 2, 2]`

- `t=0`: push all three `(2,0)`
- Pop first: `t=2`, turnaround `2-0=2`, total=2
- Pop second: `t=4`, turnaround `4-0=4`, total=6
- Pop third: `t=6`, turnaround `6-0=6`, total=12

  Average = `12/3 = 4` → **4**

Interviewer:

Good. Add a quick test harness and list expected outputs.

Candidate — Tests (expected outputs in comments)

```
def _test():
    print(min_average_turnaround([0, 1, 2], [3, 9, 6]))     # 9
    print(min_average_turnaround([0, 0, 0], [2, 2, 2]))     # 4
    print(min_average_turnaround([5], [7]))                 # 7  (finishes at 12)
    print(min_average_turnaround([0, 10, 20], [10, 1, 1])) # 4
    print(min_average_turnaround([0, 2, 4], [100, 2, 2]))   # 64
# _test()
```

Interviewer:

Great. Can you give me the complexity with reasoning?

Candidate — Complexity (with reasoning)

- **Time:**
  - We push each job **once** and pop **once** from the min-heap → `2n` heap ops total, each `O(log n)` → `O(n log n)`.
  - The inner enqueue loop advances pointer `i` monotonically; the outer loop doesn't revisit work.
  - **Overall:** `O(n log n)`.
- **Space:**
  - Heap may hold up to `n` jobs in the worst case; plus a few scalars.
  - **Overall extra space:** `O(n)`.

Interviewer (Targeted Follow-ups):

1. What happens with long idle gaps?
2. Are there any pitfalls with very large times or durations?

3. If we wanted the **actual schedule**, how would you return it?

Candidate:

1. When the heap is empty and there are still jobs left, I **jump `time`** to the next job's arrival. That prevents spinning and models idling accurately.

2. Large values are fine in Python (unbounded ints). Complexity still holds since it depends on `n`, not on numeric magnitudes.

3. I'd store `(duration, arrival, index)` in the heap and append `index` to a `schedule` list each time I pop. Return that list in addition to the average.

Interviewer — Wrap-up:
Excellent. You summarized first, asked precise clarifications, selected a **Greedy + Min-Heap** approach with a clear correctness sketch, wrote clean code, performed transparent dry runs (including detailed math for the 9 and 4 results), provided tests and a solid complexity explanation. This meets Amazon's bar.

Ratings

- Coding: **4/4**

- Problem Solving: **4/4**

- Communication: **4/4**