Amazon Interviewer:
Hi Can, welcome back. We'll do a DSA problem framed in an Amazon fulfillment context. Please think aloud. Ready?

Candidate (Excellent):
Ready. I'll summarize the problem after you present it, then ask clarifying questions and proceed.

Interviewer — Problem Statement (Amazon context)

At an Amazon fulfillment center, a picker's handheld scanner records a sequence of scanned **SKU IDs** as they walk past bins. For a specific customer order, we also have a **requirement**: each SKU in the order and how many units are needed.

- You're given:
  - `scans`: an array of SKU IDs (strings or integers) in the order they were scanned.
  - `required`: a dictionary `sku -> quantity` (all quantities ≥ 1).

Task: Find the **shortest contiguous segment** of `scans` that contains **at least** the required quantity for **every** SKU in `required`.
Return the segment as **zero-based indices** `[start, end]`. If no such segment exists, return `[-1, -1]`.

Examples

1.

`scans = ["A","D","B","C","A","B","C","A"]`
`required = {"A":2, "B":1, "C":1}`
Output: `[4, 7]`
Why: The segment `["A","B","C","A"]` (indices 4..7) contains A×2, B×1, C×1, and its length is 4.
Earlier valid segment `0..4` (`["A","D","B","C","A"]`) has length 5, so `[4,7]` is shorter.

2.

`scans = ["x","y","z"]`
`required = {"x":1, "y":1, "z":1}`
Output: `[0, 2]` (the entire array)

3.

`scans = ["p","q"]`
`required = {"p":1, "q":2}`
Output: `[-1, -1]` (not enough `q`)

Constraints

- `1 ≤ len(scans) ≤ 2 * 10^5`
- `1 ≤ len(required) ≤ 10^5`
- Each `required[sku] ≥ 1`
- IDs are hashable (string or int)

Candidate — Problem Summary:
We're given a scan sequence and a multiset of required SKUs with counts. We need the **shortest contiguous subarray** that covers all required counts (≥ per SKU). If none exists, return `[-1, -1]`. Indices are zero-based. Constraints suggest we need around **O(n)** or **O(n log n)**; given the contiguity requirement, a **linear sliding window** sounds promising.

Did I capture this correctly?

Interviewer:
Perfect.

Candidate — Clarifying Questions:

1. If `required` is empty (no items needed), should I return a zero-length segment like `[0, -1]` or `[-1, -1]`?
2. If multiple shortest segments exist, any one is acceptable?
3. `scans` can contain SKUs not in `required`; these are allowed inside the segment, correct?

Interviewer:

1. Treat `required` as non-empty for this interview (each quantity ≥ 1).

2. Yes, any shortest segment is fine.

3. Correct—extra SKUs may appear; they don't invalidate the window.

Candidate — Approach Exploration

Brute force (reject):
Enumerate all subarrays and check coverage → `O(n^2)` windows and each check involves counting → too slow for `2e5`.

Chosen Pattern (used because it fits and optimizes):
Pattern #1 — Sliding Window with two pointers and frequency maps.

Why it fits:
We're looking for a **shortest contiguous** segment that satisfies a **per-window constraint** ("window coverage ≥ required counts"). Sliding window lets us expand to reach feasibility, then **shrink greedily** to minimal length.

How it works (with the example integrated):

Example 1
`scans = ["A","D","B","C","A","B","C","A"]`
`required = {"A":2, "B":1, "C":1}`

Initialize state

- `left_index = 0`
- `best_start_index = -1`, `best_end_index = -1`, `best_window_length = +∞`
- `required_counts = {"A":2,"B":1,"C":1}`
- `required_types = 3` (distinct required SKUs)
- `window_counts = {}` (all 0)
- `satisfied_types = 0`

Expand `right_index` from 0 to end:

1. `right=0` → `"A"`
   `window_counts["A"]=1` (<2) → `satisfied_types=0` → **not feasible**.

2. `right=1` → `"D"`
   Not required → ignore for satisfaction → **not feasible**.

3. `right=2` → `"B"`
   `window_counts["B"]=1` meets B×1 → `satisfied_types=1` → **not feasible** yet.

4. `right=3` → `"C"`
   `window_counts["C"]=1` meets C×1 → `satisfied_types=2` → **not feasible** yet (A needs 2).

5. `right=4` → `"A"`
   `window_counts["A"]=2` meets A×2 → `satisfied_types=3` (= `required_types`) → **feasible window `[0..4]`**.
   Shrink from left to minimize:

   - `left=0` is `"A"` → dropping would make A `2→1` (<2) ⇒ **stop shrinking**.
     Record best: `[0,4]` (length 5).

6. `right=5` → `"B"`
   - `window_counts["B"]=2`
     Still feasible. Try shrinking:

   - `left=0` is `"A"` → A would become `2→1` ⇒ **break feasibility** ⇒ stop.
     Best remains `[0,4]`.

7. `right=6` → `"C"`
   - `window_counts["C"]=2`
     Still feasible. Try shrinking:

   - `left=0 "A"` → would break ⇒ stop.
     Best unchanged.

8. `right=7` → `"A"`
   `window_counts["A"]=3`. **Shrink aggressively** while staying feasible:

   - Drop `left=0 "A"`: A `3→2` (still ≥2) ⇒ feasible → `left=1`

   - Drop `left=1 "D"`: not required ⇒ feasible → `left=2`

   - Drop `left=2 "B"`: B `2→1` (still ≥1) ⇒ feasible → `left=3`

   - Drop `left=3 "C"`: C `2→1` (still ≥1) ⇒ feasible → `left=4`

   - Next would be dropping `left=4 "A"`: A `2→1` (**<2**) ⇒ **stop**.
     Now the minimal feasible window ending at `right=7` is **`[4,7]`** (length 4).

Update best to `[4,7]`.

Final answer: `[4,7]`.

Key idea: expand `right` to become feasible, then move `left` to trim extraneous items while keeping the window feasible, updating the best answer whenever we're feasible.

Candidate — Pseudocode

```
required_counts = frequency map from required (sku -> needed_count)
need_kinds      = number of keys in required_counts
window_counts   = empty map (sku -> count in window)
covered_kinds   = 0
left            = 0
best_start, best_end, best_length = -1, -1, +inf

for right in range(len(scans)):
    sku = scans[right]
    add sku to window_counts
    if sku in required and window_counts[sku] just reached required_counts[sku]:
        covered_kinds += 1

    while covered_kinds == need_kinds:
        # Feasible: try to shrink from the left
        if right - left + 1 < best_length:
            best_length = right - left + 1
            best_start, best_end = left, right

        left_sku = scans[left]
        remove left_sku from window_counts
        if left_sku in required and window_counts[left_sku] just dropped below
required_counts[left_sku]:
            covered_kinds -= 1
        left += 1

return [best_start, best_end]
```

Pattern Callout: Using **Sliding Window (#1)** because we need a shortest **contiguous** segment satisfying per-window frequency constraints.

Interviewer:
Looks good. Please implement with meaningful variable names and explanatory comments.

Candidate — Code (Python, descriptive names + comments)

```
from collections import Counter
from math import inf
from typing import List, Dict, Tuple, Hashable

def shortest_cover_segment(scans: List[Hashable], required: Dict[Hashable, int])
-> List[int]:
    """
    Find the shortest [start, end] segment of `scans` that covers all required
SKUs with
    at least the specified counts. Return [-1, -1] if not possible.

    Sliding Window (expand right, shrink left) with frequency maps.

    Time:  O(n) average, each index enters/leaves the window at most once.
    Space: O(U) where U is number of distinct SKUs seen in the window.
    """
    if not scans or not required:
        # Per interview, required is non-empty; returning [-1,-1] for safety if
empty.
        return [-1, -1]

    required_counts: Dict[Hashable, int] = dict(required)
    needed_kinds: int = len(required_counts)

    window_counts: Counter = Counter()
    covered_kinds: int = 0

    best_start: int = -1
    best_end: int = -1
    best_length: int = float('inf')

    left: int = 0

    for right, sku in enumerate(scans):
        # include current sku in the window
```

```
        window_counts[sku] += 1

        # If this sku is required and we just satisfied its required count,
increment covered_kinds
        if sku in required_counts and window_counts[sku] == required_counts[sku]:
            covered_kinds += 1

        # If all required kinds are covered, try to shrink from the left to get a
minimal window
        while covered_kinds == needed_kinds and left <= right:
            current_window_length = right - left + 1
            if current_window_length < best_length:
                best_length = current_window_length
                best_start = left
                best_end = right

            left_sku = scans[left]
            window_counts[left_sku] -= 1

            # If left_sku is required and we just fell below its needed count, we
lose coverage
            if left_sku in required_counts and window_counts[left_sku] <
required_counts[left_sku]:
                covered_kinds -= 1

            # Move left pointer forward to continue shrinking
            left += 1

    return [best_start, best_end]
```

Interviewer:
Great. Let's dry-run the primary example and then run a quick test harness.

Candidate — Dry Run (Example 1)

```
scans = ["A","D","B","C","A","B","C","A"]
required = {"A":2, "B":1, "C":1}
```

- Expand until `right=4` (second "A"): now A×2, B×1, C×1 → feasible at `[0..4]` (length 5).

- Shrink fails immediately because dropping the left "A" breaks A×2. Best is `[0,4]`.
- Continue expanding to the end; each time we regain plenty of A's, shrinking removes non-required "D" and older items.
- Final minimal window discovered: `[4,7]` of length 4 (`["A","B","C","A"]`).

Result: `[4, 7]`.

Candidate — Tests (expected outputs in comments)

```
def _tests():
    print(shortest_cover_segment(
        ["A","D","B","C","A","B","C","A"],
        {"A":2,"B":1,"C":1}))  # [4, 7]

    print(shortest_cover_segment(
        ["x","y","z"],
        {"x":1,"y":1,"z":1}))  # [0, 2]

    print(shortest_cover_segment(
        ["p","q"],
        {"p":1,"q":2}))        # [-1, -1]

    print(shortest_cover_segment(
        ["a","b","a","b","c"],
        {"a":1,"c":1}))        # [2, 4] or [0,4], shortest is [2,4]

    print(shortest_cover_segment(
        ["a","a","b","b","c","c"],
        {"a":2,"b":2,"c":2}))  # [0, 5] (entire array)

# _tests()
```

Interviewer:
Explain your time and space complexity with reasoning, not just big-O symbols.

Candidate — Complexity (with reasoning)

- **Time Complexity:**
  The right pointer advances from 0 to `n-1` (n steps). The left pointer only moves forward and never resets—each index is removed from the window at most once. The inner `while` loop therefore advances `left` at most `n` total times across the run. All map updates and comparisons are O(1) average (hash map).
  Total: **~O(n)** average time.

- **Space Complexity:**
  `required_counts` holds at most the number of distinct required SKUs; `window_counts` holds at most the number of distinct SKUs present in the current window. In the worst case, this is O(U), where U ≤ n.
  Total extra space: O(U) (commonly summarized as **O(n)** in the worst case, but practically bounded by the unique SKUs).

Interviewer (targeted follow-ups):

1. What if some `required` counts are larger than the total occurrences in `scans`?

2. How do you ensure we always return the **shortest** feasible window?

3. If we also needed the **count** of shortest windows (how many distinct minimal segments), how might you extend this?

Candidate:

1. The algorithm will simply **never reach** `covered_kinds == needed_kinds`, so `best_start` stays `-1` and we return `[-1, -1]`.

2. We update the best answer **only** when the window is **currently feasible**, and then we **shrink greedily** from the left as far as possible before expanding again. That produces the minimal window for each `right`, and we track the global minimum over time.

3. To count minimal segments: when we shrink a feasible window, if the new window length equals the best length, increment a counter; if the new window is shorter, reset the counter to 1 and update the best. Be careful to distinguish equal-length windows starting at different positions.

Interviewer — Wrap-up:
Excellent session. You led with a clear summary, asked precise clarifications, chose **Sliding Window (Pattern #1)** because it truly fit, articulated the feasibility/shrinking invariant, wrote clean code with meaningful variable names and comments, and provided dry runs, tests, and well-reasoned complexity. This meets Amazon's bar.

Ratings

- Coding: **4/4**
- Problem Solving: **4/4**
- Communication: **4/4**