
WEEK 1 – DAY 6

Mixed Practice, Pattern Switching, Admin Visibility

Sony uses mixed days to evaluate:

- Whether you can **identify the right pattern quickly**
- Whether you avoid over-engineering
- Whether you can connect **backend → UI → auditability**

PART A: Mixed Practice — 4 Medium Problems

These are **very realistic Sony interview questions**.

Problem 1: Longest Subarray with Sum = K

Pattern: Prefix Sum + Hashing

Interview Question

Given an array of integers and an integer k, return the **length of the longest contiguous subarray** whose sum equals k.

Spoken Explanation (What You Say Aloud)

"I use a prefix sum with a hash map. As I iterate, I keep track of the cumulative sum. If prefixSum − k has been seen before, it means the subarray between those indices sums to k. I store only the **first occurrence** of each prefix sum to maximize the subarray length."

Why This Works

- Converts subarray sum to a **difference of prefix sums**
- Hashing allows **O(1)** lookups
- Works with **negative numbers** (sliding window cannot)

Python Solution (WITH INPUT)

```
# Program: longest_subarray_sum_k.py


def longestSubarraySum(nums, k):
```

```
    prefix_sum = 0
    index_map = {0: -1}  # prefix sum -> earliest index
    max_len = 0

    for i, num in enumerate(nums):
        prefix_sum += num

        if prefix_sum - k in index_map:
            max_len = max(max_len, i - index_map[prefix_sum - k])

        if prefix_sum not in index_map:
            index_map[prefix_sum] = i

    return max_len



# INPUT
nums = [1, -1, 5, -2, 3]
k = 3
print(longestSubarraySum(nums, k))  # Output: 4
```

JavaScript Solution (WITH INPUT)

```
// Program: longestSubarraySumK.js

function longestSubarraySum(nums, k) {
    let prefixSum = 0;
    const indexMap = new Map();
    indexMap.set(0, -1);
    let maxLen = 0;

    for (let i = 0; i < nums.length; i++) {
```

```
        prefixSum += nums[i];

        if (indexMap.has(prefixSum - k)) {
            maxLen = Math.max(maxLen, i - indexMap.get(prefixSum - k));
        }

        if (!indexMap.has(prefixSum)) {
            indexMap.set(prefixSum, i);
        }
    }
    return maxLen;
}

// INPUT
const nums = [1, -1, 5, -2, 3];
const k = 3;
console.log(longestSubarraySum(nums, k)); // Output: 4
```

Complexity

- **Time:** O(n)
- **Space:** O(n)

Sony Follow-Ups

1. Why store first index only?
2. What if all numbers are positive?
3. Can sliding window work?
4. Streaming data case?

---

1 Why store the first index only?

Spoken Explanation (Say This)

“I store the first occurrence of each prefix sum because it gives the maximum possible distance to a future index, which maximizes the subarray length. If I overwrite it with a later index, I would only get shorter subarrays.”

Key Point

- First index → **longest length**
- Last index → **shorter length**

---

2 What if all numbers are positive?

Spoken Explanation

“When all numbers are positive, the subarray sum grows as the window expands and shrinks as it contracts, so a sliding window approach becomes valid and more space-efficient.”

Python (Sliding Window – WITH INPUT)

```
# Program: longest_subarray_positive_numbers.py


def longestSubarrayPositive(nums, k):
    left = 0
    curr_sum = 0
    max_len = 0

    for right in range(len(nums)):
        curr_sum += nums[right]

        while curr_sum > k:
            curr_sum -= nums[left]
            left += 1

        if curr_sum == k:
            max_len = max(max_len, right - left + 1)

    return max_len
```

```
# INPUT
nums = [1, 2, 1, 1, 1]
k = 3
print(longestSubarrayPositive(nums, k))  # Output: 3
```

JavaScript (Sliding Window – WITH INPUT)

```
// Program: longestSubarrayPositiveNumbers.js

function longestSubarrayPositive(nums, k) {
    let left = 0, sum = 0, maxLen = 0;

    for (let right = 0; right < nums.length; right++) {
        sum += nums[right];

        while (sum > k) {
            sum -= nums[left];
            left++;
        }

        if (sum === k) {
            maxLen = Math.max(maxLen, right - left + 1);
        }
    }
    return maxLen;
}

// INPUT
console.log(longestSubarrayPositive([1, 2, 1, 1, 1], 3)); // 3
```

---

3 Can sliding window work in the general case?

Spoken Explanation

“No. Sliding window fails when negative numbers exist because the sum is no longer monotonic. Expanding the window might decrease the sum, breaking the core sliding-window assumption.”

Sony Verdict

- ❌ Negative numbers → sliding window fails
- ✅ Prefix sum + hashing → always works

4️⃣ Streaming data case?

Spoken Explanation

“This problem adapts well to streaming data. I process each incoming number, update the prefix sum, and update the hash map incrementally without storing the entire array.”

Python (Streaming Style – WITH INPUT)

```
# Program: longest_subarray_streaming.py


stream = [1, -1, 5, -2, 3]

k = 3


prefix_sum = 0

index_map = {0: -1}

max_len = 0


for i, num in enumerate(stream):

    prefix_sum += num


    if prefix_sum - k in index_map:

        max_len = max(max_len, i - index_map[prefix_sum - k])


    if prefix_sum not in index_map:

        index_map[prefix_sum] = i


print(max_len)  # Output: 4
```

JavaScript (Streaming Style – WITH INPUT)

```
// Program: longestSubarrayStreaming.js

const stream = [1, -1, 5, -2, 3];
const k = 3;

let prefixSum = 0;
let maxLen = 0;
const indexMap = new Map();
indexMap.set(0, -1);

stream.forEach((num, i) => {
   prefixSum += num;

   if (indexMap.has(prefixSum - k)) {
      maxLen = Math.max(maxLen, i - indexMap.get(prefixSum - k));
   }

   if (!indexMap.has(prefixSum)) {
      indexMap.set(prefixSum, i);
   }
});

console.log(maxLen); // 4
```

Sony One-Line Summary (Memorize This)

"Storing the first prefix-sum index maximizes subarray length; sliding window works only for positive numbers, while prefix sum with hashing supports negatives and streaming data."

Problem 2: Minimum Window Substring

Pattern: Sliding Window + Hashing

Interview Question

Given two strings s and t, return the **minimum window substring of s** that contains **all characters of t** (including duplicates).
If no such window exists, return an empty string.

Spoken Explanation (What You Say Aloud)

"I use a sliding window with a hash map to track required characters. I expand the window to include all characters of t, and once the condition is satisfied, I shrink the window from the left to find the smallest valid window. This expand-and-shrink process guarantees optimality in linear time."

Key Insight (Memorize This)

"Expand the window until all required characters are satisfied, then shrink it to minimize the window size."

Python Solution (WITH INPUT)

```
# Program: minimum_window_substring.py

from collections import Counter


def minWindow(s, t):
    if not s or not t:
        return ""


    need = Counter(t)
    count = len(t)
    left = 0
    min_len = float("inf")
    start = 0


    for right in range(len(s)):
        if need[s[right]] > 0:
            count -= 1
        need[s[right]] -= 1
```

```
        while count == 0:
            if right - left + 1 < min_len:
                min_len = right - left + 1
                start = left

            need[s[left]] += 1
            if need[s[left]] > 0:
                count += 1
            left += 1

    return "" if min_len == float("inf") else s[start:start + min_len]



# INPUT
s = "ADOBECODEBANC"
t = "ABC"
print(minWindow(s, t))  # Output: "BANC"
```

JavaScript Solution (WITH INPUT)

```
// Program: minimumWindowSubstring.js
function minWindow(s, t) {
    if (!s || !t) return "";

    const need = {};
    for (let ch of t) need[ch] = (need[ch] || 0) + 1;

    let left = 0;
    let count = t.length;
    let minLen = Infinity;
    let start = 0;
```

```
    for (let right = 0; right < s.length; right++) {
        if (need[s[right]] > 0) count--;
        need[s[right]] = (need[s[right]] || 0) - 1;

        while (count === 0) {
            if (right - left + 1 < minLen) {
                minLen = right - left + 1;
                start = left;
            }

            need[s[left]]++;
            if (need[s[left]] > 0) count++;
            left++;
        }
    }

    return minLen === Infinity ? "" : s.substring(start, start + minLen);
}

// INPUT
console.log(minWindow("ADOBECODEBANC", "ABC")); // "BANC"
```

Complexity

- **Time:** O(n)
- **Space:** O(m) where m = |t|

Sony Follow-Ups

1. Why decrement even if char not needed?
2. Time complexity?
3. Unicode handling?

4. Real-time text scan?

1️⃣ Why decrement even if the character is not needed?

Spoken Answer:

"Decrementing even for non-needed characters allows me to track surplus characters. A negative count indicates extra occurrences, which lets me safely shrink the window later without breaking validity."

Key Point:

- Negative count → extra character
- Positive count → still needed

2️⃣ Time complexity?

Spoken Answer:

"Each character enters and leaves the window at most once. Both pointers move forward only, so the time complexity is linear."

- **Time:** O(n)
- **Space:** O(m) where m = |t|

3️⃣ Unicode handling?

Spoken Answer:

"This solution supports Unicode naturally because it uses hash maps rather than fixed-size arrays, so it works for any character set."

✔ Hash map → Unicode safe
✖ Fixed array (size 26/128) → not Unicode safe

4️⃣ Real-time text scan?

Spoken Answer:

"This approach works well for real-time or streaming input because it processes characters incrementally and never recomputes previous state."

Use cases Sony cares about:

- Live subtitle analysis
- Log scanning
- Real-time content filtering

Python Solution (WITH INPUT)

```
# Program: minimum_window_substring.py
from collections import Counter

def minWindow(s, t):
    if not s or not t:
        return ""

    need = Counter(t)
    count = len(t)
    left = 0
    min_len = float("inf")
    start = 0

    for right in range(len(s)):
        if need[s[right]] > 0:
            count -= 1
        need[s[right]] -= 1

        while count == 0:
            if right - left + 1 < min_len:
                min_len = right - left + 1
                start = left

            need[s[left]] += 1
            if need[s[left]] > 0:
                count += 1
            left += 1

    return "" if min_len == float("inf") else s[start:start + min_len]
```

```
# INPUT
s = "ADOBECODEBANC"
t = "ABC"
print(minWindow(s, t))  # Output: BANC
```

JavaScript Solution (WITH INPUT)

```
// Program: minimumWindowSubstring.js
function minWindow(s, t) {
    if (!s || !t) return "";

    const need = {};
    for (let ch of t) need[ch] = (need[ch] || 0) + 1;

    let left = 0;
    let count = t.length;
    let minLen = Infinity;
    let start = 0;

    for (let right = 0; right < s.length; right++) {
        if (need[s[right]] > 0) count--;
        need[s[right]] = (need[s[right]] || 0) - 1;

        while (count === 0) {
            if (right - left + 1 < minLen) {
                minLen = right - left + 1;
                start = left;
            }

            need[s[left]]++;
```

```
        if (need[s[left]] > 0) count++;
        left++;
      }
    }

    return minLen === Infinity ? "" : s.substring(start, start + minLen);
}

// INPUT
console.log(minWindow("ADOBECODEBANC", "ABC")); // BANC
```

Sony One-Line Summary (Memorize)

"Decrementing every character lets me track surplus safely, enabling optimal shrinking of the sliding window in linear time—even for Unicode and streaming data."

Problem 3: Trapping Rain Water

Pattern: Two Pointers

Spoken Explanation (What You Say Aloud)

"Water trapped at any index depends on the minimum of the maximum height to its left and right. Using two pointers, I maintain running left-max and right-max values and compute trapped water in one pass with constant space."

Key Insight (Say This Clearly)

"The smaller side determines the water level, so I always move the pointer with the smaller height."

Python Solution (WITH INPUT)

```
# Program: trapping_rain_water.py

def trap(height):
    if not height:
        return 0
```

```
    left, right = 0, len(height) - 1
    left_max = right_max = 0
    water = 0

    while left < right:
        if height[left] < height[right]:
            left_max = max(left_max, height[left])
            water += left_max - height[left]
            left += 1
        else:
            right_max = max(right_max, height[right])
            water += right_max - height[right]
            right -= 1

    return water


# INPUT
heights = [0,1,0,2,1,0,1,3,2,1,2,1]
print(trap(heights))  # Output: 6
```

JavaScript Solution (WITH INPUT)

```
// Program: trappingRainWater.js

function trap(height) {
    if (height.length === 0) return 0;

    let left = 0, right = height.length - 1;
    let leftMax = 0, rightMax = 0;
    let water = 0;
```

```
    while (left < right) {
        if (height[left] < height[right]) {
            leftMax = Math.max(leftMax, height[left]);
            water += leftMax - height[left];
            left++;
        } else {
            rightMax = Math.max(rightMax, height[right]);
            water += rightMax - height[right];
            right--;
        }
    }
    return water;
}

// INPUT
console.log(trap([0,1,0,2,1,0,1,3,2,1,2,1])); // Output: 6
```

Complexity

- **Time:** O(n)
- **Space:** O(1)

Sony Follow-Ups

1. Why two pointers instead of prefix arrays?
2. Space trade-off?
3. Can this be parallelized?

1️⃣ Why two pointers instead of prefix arrays?

Spoken Answer:

“Prefix arrays require extra memory for left-max and right-max arrays. Two pointers compute the same result in a single pass using constant space.”

Comparison:

Approach	Time	Space
Prefix Arrays	O(n)	O(n)
Two Pointers	O(n)	O(1) ✅

---

2 Space trade-off?

Spoken Answer:

"Two pointers trade a bit of reasoning complexity for significant space savings, which is preferred in performance-critical systems."

Sony preference: **Lower memory footprint**

---

3 Can this be parallelized?

Spoken Answer:

"The two-pointer approach itself is sequential. However, prefix max arrays can be computed in parallel, then combined, at the cost of additional space."

Real-world takeaway:

- Parallel → prefix arrays
- Optimal memory → two pointers

Sony One-Line Summary (Memorize)

"Two pointers work because water level is limited by the smaller boundary, allowing linear-time computation with constant space."

Problem 4: Top K Frequent Elements

Pattern: Hashing + Heap

Question

Return the k most frequent elements.

Spoken Explanation (What You Say Aloud)

"I first count the frequency of each element using hashing. Then I use a heap to efficiently extract the k elements with the highest frequency without sorting the entire dataset."

Key Insight (Say This Clearly)

“A heap allows us to get top-k elements in O(n log k) time instead of sorting all frequencies.”

Python Solution (WITH INPUT)

```
# Program: top_k_frequent_elements.py

from collections import Counter
import heapq

def topKFrequent(nums, k):
    freq = Counter(nums)
    return heapq.nlargest(k, freq.keys(), key=freq.get)

# INPUT
nums = [1, 1, 1, 2, 2, 3, 3, 3, 3, 4]
k = 2
print(topKFrequent(nums, k))  # Output: [3, 1]
```

JavaScript Solution (WITH INPUT)

```
// Program: topKFrequentElements.js

function topKFrequent(nums, k) {
    const freq = new Map();

    for (let num of nums) {
        freq.set(num, (freq.get(num) || 0) + 1);
    }

    // Convert map entries to array and sort by frequency
    const sorted = Array.from(freq.entries())
                        .sort((a, b) => b[1] - a[1]);
```

```
    return sorted.slice(0, k).map(item => item[0]);
}

// INPUT
console.log(topKFrequent([1,1,1,2,2,3,3,3,3,4], 2)); // Output: [3, 1]
```

Complexity

- **Time:**
  - Heap approach: O(n log k)
  - JS sorting approach: O(n log n)
- **Space:** O(n)

Sony Follow-Ups

1. Heap vs bucket sort?
2. Streaming numbers?
3. Memory optimization?

---

1 Heap vs Bucket Sort?

Spoken Answer:

"Heap is better when k is small compared to n. Bucket sort achieves linear time but uses more memory."

Approach	Time	Space	When to Use
Heap	O(n log k)	O(n)	Small k
Bucket Sort	O(n)	O(n)	Large datasets

---

2 Streaming Numbers?

Spoken Answer:

"For streaming data, we maintain a frequency map and a min-heap of size k, updating it as new elements arrive."

Why Sony likes this:

- Works for real-time analytics
- Memory-bounded

3️⃣ Memory Optimization?

Spoken Answer:

"If the value range is small, we can use arrays instead of hash maps. For streams, we cap heap size to k."

Additional Optimization:

- Use **Count-Min Sketch** for approximate frequencies in very large streams

---

Sony One-Line Summary (Memorize)

"Hashing counts frequencies and a heap extracts the top-k efficiently without full sorting."

PART B: Project — Admin Audit Log UI (Basic Table)

Interview Framing

"Design a simple admin UI to view audit logs."

Sony is **not testing design polish**, but:

- Data clarity
- Pagination thinking
- Security awareness

Minimum Fields to Display

- Timestamp
- User ID
- Action
- Resource
- Status

Option 1: Simple HTML Table (Backend-Rendered)

```
<table border="1">
  <thead>
```

```
    <tr>
      <th>Timestamp</th>
      <th>User ID</th>
      <th>Action</th>
    </tr>
  </thead>
  <tbody>
    {% for log in logs %}
    <tr>
      <td>{{ log.timestamp }}</td>
      <td>{{ log.user_id }}</td>
      <td>{{ log.action }}</td>
    </tr>
    {% endfor %}
  </tbody>
</table>
```

Option 2: React Basic Table (Sony-Friendly)

```
function AuditTable({ logs }) {
  return (
    <table>
      <thead>
        <tr>
          <th>Time</th>
          <th>User</th>
          <th>Action</th>
        </tr>
      </thead>
      <tbody>
        {logs.map(log => (
          <tr key={log.id}>
```

```
          <td>{log.timestamp}</td>
          <td>{log.user_id}</td>
          <td>{log.action}</td>
        </tr>
      ))}
    </tbody>
  </table>
 );
}
```

Backend API (Example)

```
@app.route("/admin/audit-logs")
def get_logs():
  logs = AuditLog.query.order_by(AuditLog.timestamp.desc()).limit(100)
  return jsonify([{
    "id": l.id,
    "user_id": l.user_id,
    "action": l.action,
    "timestamp": l.timestamp
  } for l in logs])
```

What Sony Wants You to Say

“Admin access is role-protected, logs are read-only, paginated, and sorted by timestamp.”

Sony Follow-Ups on UI

1. How do you restrict admin access?
2. Pagination strategy?
3. Filtering by user or date?
4. How to handle millions of logs?
5. Should logs be editable? (Correct answer: No)
