WEEK 1 – DAY 4

Sliding Window, Real-Time Thinking, API Flow Design

Sony evaluates:

- Whether you can **maintain state efficiently**
- Whether you avoid recomputation
- Whether you can **model request flows clearly**
- Whether you think in **production pipelines**, not just code

---

1. Sliding Window – Core Concept

What You Should Say (Spoken)

"Sliding window is an optimization technique where we maintain a dynamic window over a sequence and update the window state incrementally, instead of recomputing results from scratch for every subarray or substring."

When Sony Expects Sliding Window (Spoken)

"Sony expects sliding window when working with substrings or subarrays, continuous or streaming data, fixed or variable window sizes, and performance-critical paths where brute force is too slow."

Sliding Window Types (Say This)

Fixed size	Maximum sum subarray of size K
Variable size	Longest substring without repeating characters

Problem 1: Fixed Window

Maximum Sum Subarray of Size K

Program / File Name

max_sum_subarray_k.py
maxSumSubarrayK.js

Interview Question

Given an array and an integer k, find the maximum sum of any contiguous subarray of size k.

Key Insight (Say This)

“Since the window size is fixed, I add the incoming element and remove the outgoing element in constant time.”

Python (With Input)

```
nums = [2, 1, 5, 1, 3, 2]
k = 3


def maxSumSubarray(nums, k):
    window_sum = sum(nums[:k])
    max_sum = window_sum


    for i in range(k, len(nums)):
        window_sum += nums[i] - nums[i - k]
        max_sum = max(max_sum, window_sum)


    return max_sum


print(maxSumSubarray(nums, k))  # Output: 9
```

JavaScript (With Input)

```
const nums = [2, 1, 5, 1, 3, 2];
const k = 3;


function maxSumSubarray(nums, k) {
    let windowSum = nums.slice(0, k).reduce((a, b) => a + b, 0);
    let maxSum = windowSum;
    for (let i = k; i < nums.length; i++) {
        windowSum += nums[i] - nums[i - k];
        maxSum = Math.max(maxSum, windowSum);
    }
    return maxSum;
}

console.log(maxSumSubarray(nums, k)); // 9
```

Complexity (Say This)

“Time complexity is O(n) and space complexity is O(1).”

Problem 2: Variable Window

Longest Substring Without Repeating Characters

Interview Question

Given a string, find the length of the longest substring without repeating characters.

Key Insight (Say This)

“I expand the window until it becomes invalid, then shrink it from the left until it becomes valid again.”

Python (With Input)

```
s = "abcabcbb"

def lengthOfLongestSubstring(s):
    char_set = set()
    left = 0
    max_len = 0
    for right in range(len(s)):
        while s[right] in char_set:
            char_set.remove(s[left])
            left += 1
        char_set.add(s[right])
        max_len = max(max_len, right - left + 1)

    return max_len

print(lengthOfLongestSubstring(s))  # Output: 3
```

JavaScript (With Input)

```
const s = "abcabcbb";

function lengthOfLongestSubstring(s) {
    let set = new Set();
    let left = 0, maxLen = 0;

    for (let right = 0; right < s.length; right++) {
        while (set.has(s[right])) {
            set.delete(s[left]);
            left++;
        }
        set.add(s[right]);
        maxLen = Math.max(maxLen, right - left + 1);
    }
    return maxLen;
}
```

```
console.log(lengthOfLongestSubstring(s)); // 3
```

Complexity (Say This)

“Each character is added and removed at most once, so time complexity is O(n) and space is O(n).”

Interview Warm-Up Questions

1. Why sliding window over brute force?
2. Difference between sliding window and two pointers?
3. Can sliding window work with negative numbers?
4. What breaks sliding window logic?
5. How do you track window state?

1️⃣ Why sliding window over brute force?

Spoken

“Brute force recalculates overlapping subarrays repeatedly, leading to O(n^2). Sliding window reuses previous computations and reduces complexity to O(n).”

2️⃣ Difference between sliding window and two pointers?

Spoken

“Sliding window is a specialized form of two pointers where both pointers move forward to maintain a valid window and track window state.”

3️⃣ Can sliding window work with negative numbers?

Spoken

“Fixed-size sliding windows work with negative numbers, but variable-size windows relying on monotonic growth often break with negatives.”

4️⃣ What breaks sliding window logic?

Spoken

“Sliding window breaks when adding elements does not move the window closer to a valid state, such as when sums fluctuate due to negative values.”

---

5 How do you track window state?

Spoken

“I track window state using variables like sum, frequency maps, or sets that update as elements enter and leave the window.”

Sony Follow-Up Mapping (Very Important)

Question	Expected Answer
Negative numbers	Use fixed window or prefix sum
Streaming data	Sliding window is ideal
Large input	Linear scan + constant updates
Memory constraints	Prefer counters over arrays

Sony One-Line Summary (Memorize)

“Sliding window converts repeated subarray or substring computation into a linear pass by maintaining and updating window state efficiently.”

---

2. Problem 1: Longest Substring Without Repeating Characters

Interview Question

Given a string, find the length of the longest substring without repeating characters.

Brute Force (Say Briefly, Then Reject)

Spoken

“A brute-force approach would check all possible substrings and verify uniqueness, which leads to $O(n^2)$ time complexity and is not suitable for large inputs.”

Optimal Approach: Sliding Window + Hash Map

Key Insight (VERY IMPORTANT – Say This Clearly)

“When a character repeats, the window becomes invalid. To fix it efficiently, I move the left pointer to one position after the character’s previous occurrence, ensuring the substring remains unique.”

Sony interviewers listen closely to this sentence.

Spoken Explanation (Step-by-Step)

“I use a sliding window with two pointers.
The right pointer expands the window, and a hash map stores the most recent index of each character.
If a character repeats inside the current window, I move the left pointer to the index just after its previous occurrence.
This guarantees that the window always contains unique characters and each index is processed once.”

Python Solution (WITH INPUT)

Program name: longest_unique_substring.py

```
s = "abcabcbb"   # INPUT


def lengthOfLongestSubstring(s):
    char_index = {}
    left = 0
    max_len = 0


    for right in range(len(s)):
        if s[right] in char_index and char_index[s[right]] >= left:
            left = char_index[s[right]] + 1


        char_index[s[right]] = right
        max_len = max(max_len, right - left + 1)


    return max_len


print(lengthOfLongestSubstring(s))  # Output: 3
```

JavaScript Solution (WITH INPUT)

Program name: longestUniqueSubstring.js

```
const s = "abcabcbb"; // INPUT
```

```
function lengthOfLongestSubstring(s) {
   const map = new Map();
   let left = 0;
   let maxLen = 0;

   for (let right = 0; right < s.length; right++) {
      if (map.has(s[right]) && map.get(s[right]) >= left) {
         left = map.get(s[right]) + 1;
      }

      map.set(s[right], right);
      maxLen = Math.max(maxLen, right - left + 1);
   }
   return maxLen;
}

console.log(lengthOfLongestSubstring(s)); // Output: 3
```

Complexity (Say Confidently)

"Time complexity is O(n) because each character is processed once.
Space complexity is O(min(n, charset size)) due to the hash map."

Sony Follow-Ups

1. Why move left to lastIndex + 1?
2. Can this be done using a Set?
3. How would you return the substring itself?
4. How does this behave for Unicode?
5. Real-time text stream handling?

---

1 Why move left to lastIndex + 1?

Spoken Explanation (Say This)

“Moving the left pointer to lastIndex + 1 ensures the duplicate character is fully removed from the current window in one step. Any smaller move would still keep the duplicate, and moving further would unnecessarily reduce the window.”

Code Demonstration

Program Name

- Python: longest_substring_left_jump.py
- JavaScript: longestSubstringLeftJump.js

Python

```python
s = "abba"  # INPUT


def longestSubstring(s):
    last_seen = {}
    left = 0
    max_len = 0


    for right in range(len(s)):
        if s[right] in last_seen and last_seen[s[right]] >= left:
            left = last_seen[s[right]] + 1


        last_seen[s[right]] = right
        max_len = max(max_len, right - left + 1)


    return max_len


print(longestSubstring(s))  # Output: 2
```

JavaScript

```javascript
const s = "abba"; // INPUT


function longestSubstring(s) {
    const map = new Map();
    let left = 0, maxLen = 0;
    for (let right = 0; right < s.length; right++) {
        if (map.has(s[right]) && map.get(s[right]) >= left) {
            left = map.get(s[right]) + 1;
        }
        map.set(s[right], right);
        maxLen = Math.max(maxLen, right - left + 1);
    }
    return maxLen;
}

console.log(longestSubstring(s)); // 2
```

---

2 Can this be done using a Set?

Spoken Explanation

"Yes, a Set can track uniqueness, but removing characters one by one makes it slightly less efficient. A hash map is preferred because it allows jumping the left pointer directly."

Code Using Set

Program Name

- Python: longest_substring_using_set.py
- JavaScript: longestSubstringUsingSet.js

Python

```
s = "abcabcbb"  # INPUT


def longestSubstring(s):
    seen = set()
    left = 0
    max_len = 0


    for right in range(len(s)):
        while s[right] in seen:
            seen.remove(s[left])
            left += 1

        seen.add(s[right])
        max_len = max(max_len, right - left + 1)

    return max_len

print(longestSubstring(s))  # Output: 3
```

JavaScript

```
const s = "abcabcbb"; // INPUT

function longestSubstring(s) {
    const set = new Set();
    let left = 0, maxLen = 0;

    for (let right = 0; right < s.length; right++) {
        while (set.has(s[right])) {
            set.delete(s[left]);
            left++;
        }
        set.add(s[right]);
        maxLen = Math.max(maxLen, right - left + 1);
    }
    return maxLen;
}

console.log(longestSubstring(s)); // 3
```

3️⃣ How would you return the substring itself?

Spoken Explanation

“I track the starting index whenever a new maximum length is found and return the substring using that start index and length.”

Code Returning Substring

Program Name

- Python: longest_unique_substring_return_string.py
- JavaScript: longestUniqueSubstringReturnString.js

Python

```
s = "pwwkew"  # INPUT


def longestSubstring(s):
    last_seen = {}
    left = 0
    max_len = 0
    start = 0

    for right in range(len(s)):
        if s[right] in last_seen and last_seen[s[right]] >= left:
            left = last_seen[s[right]] + 1

        last_seen[s[right]] = right

        if right - left + 1 > max_len:
            max_len = right - left + 1
            start = left

    return s[start:start + max_len]

print(longestSubstring(s))  # Output: "wke"
```

JavaScript

```
const s = "pwwkew"; // INPUT
```

```
function longestSubstring(s) {
    const map = new Map();
    let left = 0, maxLen = 0, start = 0;

    for (let right = 0; right < s.length; right++) {
        if (map.has(s[right]) && map.get(s[right]) >= left) {
            left = map.get(s[right]) + 1;
        }
        map.set(s[right], right);

        if (right - left + 1 > maxLen) {
            maxLen = right - left + 1;
            start = left;
        }
    }
    return s.substring(start, start + maxLen);
}

console.log(longestSubstring(s)); // "wke"
```

---

4 How does this behave for Unicode?

Spoken Explanation

“The algorithm still works, but Unicode characters may occupy multiple code units. For full Unicode correctness, we must iterate by grapheme clusters instead of code units.”

Code Note (Interview-Level)

Program Name

- Python: unicode_longest_substring_note.py
- JavaScript: unicodeLongestSubstringNote.js

Interviewers expect **conceptual awareness**, not full Unicode parsing logic.

---

5 Real-Time Text Stream Handling?

Spoken Explanation

“Sliding window is ideal for streaming. We process one character at a time, updating the window state without storing the entire string.”

Streaming Code Simulation

Program Name

- Python: longest_substring_stream.py
- JavaScript: longestSubstringStream.js

Python

```python
stream = "abcabcbb"  # INPUT


seen = {}

left = 0

max_len = 0


for i, ch in enumerate(stream):

    if ch in seen and seen[ch] >= left:

        left = seen[ch] + 1

    seen[ch] = i

    max_len = max(max_len, i - left + 1)


print(max_len)  # Output: 3
```

JavaScript

```javascript
const stream = "abcabcbb"; // INPUT


let map = new Map();

let left = 0, maxLen = 0;


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

    if (map.has(stream[i]) && map.get(stream[i]) >= left) {

        left = map.get(stream[i]) + 1;

    }
```

```
  map.set(stream[i], i);
  maxLen = Math.max(maxLen, i - left + 1);
}

console.log(maxLen); // 3
```

Sony Final One-Line Summary (Very Strong)

"This problem is optimally solved using a sliding window with a hash map, enabling O(n) time, constant updates, and seamless support for streaming data."

---

3. Problem 2: Maximum Sum Subarray of Size K

Interview Question

Given an array of integers and a number k, find the maximum sum of any contiguous subarray of size k.

Brute Force (Say Briefly)

Spoken Explanation

"A brute-force approach would calculate the sum of every subarray of size k, which takes O(n·k) time and is inefficient for large inputs."

Optimal Approach: Fixed Sliding Window

Key Insight (VERY IMPORTANT – Say This)

"Instead of recomputing the sum for each subarray, I slide the window by adding the new element entering the window and removing the element leaving it."

Spoken Explanation (Step-by-Step)

"I first compute the sum of the first k elements.
Then I slide the window one index at a time, updating the sum by adding the next element and subtracting the element that leaves the window.
At each step, I track the maximum sum.
This allows the solution to run in linear time using constant space."

Python Solution (WITH INPUT)

Program name: max_sum_subarray_size_k.py

```python
nums = [2, 1, 5, 1, 3, 2]  # INPUT
k = 3

def maxSumSubarray(nums, k):
    if k > len(nums):
        return None

    window_sum = sum(nums[:k])
    max_sum = window_sum

    for i in range(k, len(nums)):
        window_sum += nums[i]
        window_sum -= nums[i - k]
        max_sum = max(max_sum, window_sum)

    return max_sum

print(maxSumSubarray(nums, k))  # Output: 9
```

JavaScript Solution (WITH INPUT)

Program name: maxSumSubarraySizeK.js

```javascript
const nums = [2, 1, 5, 1, 3, 2]; // INPUT
const k = 3;

function maxSumSubarray(nums, k) {
    if (k > nums.length) return null;

    let windowSum = 0;
    for (let i = 0; i < k; i++) {
        windowSum += nums[i];
    }
    let maxSum = windowSum;


    for (let i = k; i < nums.length; i++) {
        windowSum += nums[i];
        windowSum -= nums[i - k];
        maxSum = Math.max(maxSum, windowSum);
    }
    return maxSum;
}


console.log(maxSumSubarray(nums, k)); // Output: 9
```

Complexity (Say Confidently)

"Time complexity is O(n) since each element is processed once.
Space complexity is O(1) because no extra data structures are used."

Sony Follow-Ups

1. What if k > n?
2. Can this handle negative numbers?
3. How would you process a data stream?
4. Sliding window vs prefix sum?
5. Can you parallelize this?

---

1 What if k > n?

Spoken (What to Say)

"If k is larger than the array length, a contiguous subarray of size k cannot exist. I handle this as an edge case and return null or an error before processing."

Code (WITH INPUT)

Program names

- Python: max_sum_subarray_k_gt_n.py
- JavaScript: maxSumSubarrayKGreaterThanN.js

Python

```python
nums = [2, 1, 5]  # INPUT
k = 5

def maxSumSubarray(nums, k):
    if k > len(nums):
        return None
    return sum(nums[:k])

print(maxSumSubarray(nums, k))  # Output: None
```

JavaScript

```javascript
const nums = [2, 1, 5]; // INPUT
const k = 5;

function maxSumSubarray(nums, k) {
    if (k > nums.length) return null;
    return nums.slice(0, k).reduce((a, b) => a + b, 0);
}

console.log(maxSumSubarray(nums, k)); // null
```

---

2 Can this handle negative numbers?

Spoken

“Yes. Because the window size is fixed, negative values don’t affect correctness. Every window of size k is still evaluated.”

Code (WITH INPUT)

Program names

- Python: max_sum_subarray_with_negatives.py
- JavaScript: maxSumSubarrayWithNegatives.js

Python

```python
nums = [-2, -1, -3, -4]  # INPUT
```

```
k = 2


def maxSumSubarray(nums, k):
    window_sum = sum(nums[:k])
    max_sum = window_sum
    for i in range(k, len(nums)):
        window_sum += nums[i] - nums[i - k]
        max_sum = max(max_sum, window_sum)
    return max_sum


print(maxSumSubarray(nums, k))  # Output: -3
```

JavaScript

```
const nums = [-2, -1, -3, -4]; // INPUT
const k = 2;


function maxSumSubarray(nums, k) {
    let windowSum = nums[0] + nums[1];
    let maxSum = windowSum;
    for (let i = k; i < nums.length; i++) {
        windowSum += nums[i] - nums[i - k];
        maxSum = Math.max(maxSum, windowSum);
    }
    return maxSum;
}


console.log(maxSumSubarray(nums, k)); // -3
```

---

3 How would you process a data stream?

Spoken

"I maintain a rolling sum and keep only the last k elements using a queue or circular buffer, updating the sum as new data arrives."

Code (WITH INPUT)

Program names

- Python: max_sum_subarray_stream.py
- JavaScript: maxSumSubarrayStream.js

Python

```
from collections import deque


stream = [2, 1, 5, 1, 3, 2]  # INPUT
k = 3


window = deque()
window_sum = 0
max_sum = float('-inf')


for num in stream:
    window.append(num)
    window_sum += num
    if len(window) > k:
        window_sum -= window.popleft()
    if len(window) == k:
        max_sum = max(max_sum, window_sum)


print(max_sum)  # Output: 9
```

JavaScript

```
const stream = [2, 1, 5, 1, 3, 2]; // INPUT
const k = 3;


let queue = [];
let windowSum = 0;
let maxSum = -Infinity;
```

```
for (const num of stream) {
   queue.push(num);
   windowSum += num;
   if (queue.length > k) {
      windowSum -= queue.shift();
   }
   if (queue.length === k) {
      maxSum = Math.max(maxSum, windowSum);
   }
}

console.log(maxSum); // 9
```

---

4 Sliding window vs prefix sum?

Spoken

“Sliding window is ideal for fixed-size subarrays because it runs in O(n) time with O(1) space. Prefix sums use O(n) extra memory and are better when many range queries are required.”

Code (Prefix Sum Demo)

Program names

- Python: prefix_sum_range_query_demo.py
- JavaScript: prefixSumRangeQueryDemo.js

Python

```
nums = [2, 1, 5, 1, 3, 2]  # INPUT
prefix = [0]
for n in nums:
   prefix.append(prefix[-1] + n)

# sum of range [1..3]
print(prefix[4] - prefix[1])  # Output: 7
```

JavaScript

```
const nums = [2, 1, 5, 1, 3, 2]; // INPUT
const prefix = [0];

for (const n of nums) {
  prefix.push(prefix[prefix.length - 1] + n);
}

console.log(prefix[4] - prefix[1]); // 7
```

---

5 Can you parallelize this?

Spoken

"The sliding window approach is inherently sequential because each window depends on the previous one. However, prefix sum preprocessing can be parallelized."

Program Names (Conceptual)

- Python: parallel_prefix_sum_concept.py
- JavaScript: parallelPrefixSumConcept.js

(Conceptual explanation is sufficient for interviews.)

Sony One-Line Summary (Memorize)

"For fixed-size subarray problems, sliding window is optimal due to linear time, constant space, and easy adaptation to streaming data.".

---

4. Sony-Level Variations (Very Likely)

Expect **at least one**:

1. Minimum window substring
2. Longest substring with at most K distinct characters
3. Average of subarrays of size K
4. Find all anagrams in a string
5. Maximum number of vowels in substring of size K

---

1 Minimum Window Substring (HARD)

Spoken (Say This)

“This is a variable-size sliding window problem where I expand the window until all required characters are covered, then shrink it to find the minimum valid window.”

Key Insight

Maintain:

- Frequency map of t
- Count of matched characters

Program Names

- Python: minimum_window_substring.py
- JavaScript: minimumWindowSubstring.js

Python (WITH INPUT)

```
s = "ADOBECODEBANC"

t = "ABC"


def minWindow(s, t):

    from collections import Counter

    need = Counter(t)

    left = count = 0

    min_len = float('inf')

    start = 0


    for right, ch in enumerate(s):

        if ch in need:

            need[ch] -= 1

            if need[ch] >= 0:

                count += 1


        while count == len(t):

            if right - left + 1 < min_len:

                min_len = right - left + 1

                start = left
            if s[left] in need:
                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]

print(minWindow(s, t))  # Output: "BANC"
```

JavaScript

```
const s = "ADOBECODEBANC";
const t = "ABC";

function minWindow(s, t) {
    const need = {};
    for (let c of t) need[c] = (need[c] || 0) + 1;

    let left = 0, count = 0, minLen = Infinity, start = 0;

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

        while (count === t.length) {
            if (right - left + 1 < minLen) {
                minLen = right - left + 1;
                start = left;
            }
            if (need[s[left]] !== undefined) {
                if (++need[s[left]] > 0) count--;
            }
            left++;
        }
    }
    return minLen === Infinity ? "" : s.slice(start, start + minLen);
}

console.log(minWindow(s, t)); // BANC
```

---

2 Longest Substring with At Most K Distinct Characters

Spoken

“I expand the window and track distinct characters. If distinct count exceeds K, I shrink the window.”

Program Names

- Python: longest_substring_k_distinct.py
- JavaScript: longestSubstringKDistinct.js

Python

```
s = "eceba"
k = 2

def longestKDistinct(s, k):
    from collections import defaultdict
    freq = defaultdict(int)
    left = max_len = 0

    for right in range(len(s)):
        freq[s[right]] += 1
        while len(freq) > k:
            freq[s[left]] -= 1
            if freq[s[left]] == 0:
                del freq[s[left]]
            left += 1
        max_len = max(max_len, right - left + 1)

    return max_len

print(longestKDistinct(s, k))  # Output: 3
```

JavaScript

```
const s = "eceba";
const k = 2;

function longestKDistinct(s, k) {
    let map = new Map();
    let left = 0, maxLen = 0;

    for (let right = 0; right < s.length; right++) {
        map.set(s[right], (map.get(s[right]) || 0) + 1);

        while (map.size > k) {
            map.set(s[left], map.get(s[left]) - 1);
            if (map.get(s[left]) === 0) map.delete(s[left]);
            left++;
        }
        maxLen = Math.max(maxLen, right - left + 1);
    }
    return maxLen;
}

console.log(longestKDistinct(s, k)); // 3
```

---

3 Average of Subarrays of Size K (FIXED WINDOW)

Spoken

“This is a fixed-size sliding window where I update the sum incrementally and compute averages.”

Program Names

- Python: average_subarrays_size_k.py
- JavaScript: averageSubarraysSizeK.js

Python

```
nums = [1, 3, 2, 6, -1, 4, 1, 8, 2]
k = 5

def averages(nums, k):
    res = []
    window_sum = sum(nums[:k])
    res.append(window_sum / k)

    for i in range(k, len(nums)):
        window_sum += nums[i] - nums[i - k]
        res.append(window_sum / k)

    return res

print(averages(nums, k))
```

JavaScript

```
const nums = [1,3,2,6,-1,4,1,8,2];
const k = 5;

function averages(nums, k) {
    let res = [];
    let sum = nums.slice(0, k).reduce((a,b)=>a+b,0);
    res.push(sum / k);

    for (let i = k; i < nums.length; i++) {
        sum += nums[i] - nums[i - k];
        res.push(sum / k);
```

```
  }
  return res;
}

console.log(averages(nums, k));
```

---

4 Find All Anagrams in a String

Spoken

“I use a fixed sliding window with frequency comparison to detect anagrams.”

Program Names

- Python: find_all_anagrams.py
- JavaScript: findAllAnagrams.js

Python

```
s = "cbaebabacd"
p = "abc"


def findAnagrams(s, p):
  from collections import Counter
  res = []
  p_count = Counter(p)
  window = Counter()

  for i in range(len(s)):
    window[s[i]] += 1
    if i >= len(p):
      if window[s[i - len(p)]] == 1:
        del window[s[i - len(p)]]
      else:
        window[s[i - len(p)]] -= 1
    if window == p_count:
      res.append(i - len(p) + 1)
    return res

print(findAnagrams(s, p))  # [0, 6]
```

JavaScript

```
const s = "cbaebabacd";
const p = "abc";

function findAnagrams(s, p) {
    const res = [];
    const freq = {};
    for (let c of p) freq[c] = (freq[c] || 0) + 1;

    let left = 0, count = p.length;

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

        if (right - left + 1 === p.length) {
            if (count === 0) res.push(left);
            if (freq[s[left]]++ >= 0) count++;
            left++;
        }
    }
    return res;
}

console.log(findAnagrams(s, p)); // [0,6]
```

---

5 Maximum Number of Vowels in Substring of Size K

Spoken

“This is a fixed-size sliding window where I track vowel count.”

Program Names

- Python: max_vowels_substring_k.py
- JavaScript: maxVowelsSubstringK.js

Python

```
s = "abciiidef"
k = 3


def maxVowels(s, k):
    vowels = set("aeiou")
    count = sum(1 for c in s[:k] if c in vowels)
    max_count = count


    for i in range(k, len(s)):
        if s[i] in vowels: count += 1
        if s[i - k] in vowels: count -= 1
        max_count = max(max_count, count)


    return max_count


print(maxVowels(s, k))  # Output: 3
```

JavaScript

```
const s = "abciiidef";
const k = 3;


function maxVowels(s, k) {
    const vowels = new Set(['a','e','i','o','u']);
    let count = 0;


    for (let i = 0; i < k; i++)
        if (vowels.has(s[i])) count++;
    let max = count;

    for (let i = k; i < s.length; i++) {
        if (vowels.has(s[i])) count++;
        if (vowels.has(s[i - k])) count--;
        max = Math.max(max, count);
    }
    return max;
}

console.log(maxVowels(s, k)); // 3
```

Sony Final Tip (VERY IMPORTANT)

If Sony asks:

"How do you identify sliding window problems quickly?"

You say:

"If the problem involves contiguous data and asks for max, min, count, or length, sliding window is usually the optimal approach.

---

5. Project (Concept): API Flow Diagram

Auth → Action → Audit Log

Interview Framing

"Design a secure API request flow with audit logging."

High-Level Flow

Client

↓

API Gateway

↓

Auth Service (JWT / Session)

↓

Authorization Check (RBAC)

↓

Business Action

↓

Audit Log Writer

↓

Response

Step-by-Step Explanation (Say This)

1. **Authentication**
   - Validate token/session
   - Extract user_id
2. **Authorization**
   - Check user permissions
   - Enforce RBAC / ABAC
3. **Action Execution**
   - Perform requested operation
   - Capture metadata
4. **Audit Logging**
   - Asynchronous write
   - Append-only
   - Include request ID
5. **Response**
   - Return result
   - Correlate with log ID

What Sony Listens For

- Separation of concerns
- Async logging
- Failure handling

- Traceability (request_id)
- Security boundaries

Interview Follow-Ups

1. What if logging fails?
2. Should logs block the request?
3. How do you avoid duplicate logs?
4. How to trace a request across services?
5. How do you handle PII in logs?

---

6. How to Explain in Interview (Script)

“I’d ensure authentication and authorization are completed before business logic. Audit logging would be asynchronous and non-blocking, with request IDs for traceability.”
