Sliding Window Technique — JavaScript Interview Guide

An interview-focused deep dive into recognizing, implementing, and explaining sliding window problems using JavaScript.

1. What Is the Sliding Window Technique?

Sliding Window is an optimization technique used to reduce nested loops over contiguous data (arrays or strings) into a single linear pass. Instead of recomputing values for every subarray or substring, we maintain a window defined by two pointers that expands and contracts.

Two Core Variants:
• Fixed-size window (window length stays constant)
• Dynamic-size window (window expands and shrinks based on a condition)

Why interviewers like it: It tests your ability to optimize brute-force solutions, maintain invariants, and reason about time complexity.

2. How to Recognize a Sliding Window Problem

You should immediately consider sliding window when you see:
• Arrays or strings
• Contiguous subarrays or substrings
• Phrases like 'longest', 'shortest', 'maximum', 'minimum'
• A brute-force solution that would be $O(n^2)$

Interview heuristic: If both pointers only move forward and never reset, you are probably looking at an O(n) sliding window solution.

3. Common Sliding Window Gotchas (Interview Traps)

• Forgetting to update state when moving the left pointer
• Shrinking the window too aggressively
• Recomputing max/min inside the loop (breaking O(n))
• Assuming window size must always shrink when invalid

4. Best Time to Buy and Sell Stock

Goal: Maximize profit by choosing one buy day and one sell day. This is a dynamic sliding window where the left pointer tracks the minimum price seen so far.

```
function maxProfit(prices) {
    let minPrice = prices[0];
    let maxProfit = 0;

    for (let i = 1; i < prices.length; i++) {
        maxProfit = Math.max(maxProfit, prices[i] - minPrice);
        minPrice = Math.min(minPrice, prices[i]);
    }

    return maxProfit;
}
```

Interview insight: This is conceptually a sliding window where the left side is always the lowest price seen so far.

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

5. Longest Substring Without Repeating Characters

Key difficulty: handling the left pointer correctly. You must never move it backward.

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

    for (let right = 0; right < s.length; right++) {
        if (seen.has(s[right])) {
            left = Math.max(left, seen.get(s[right]) + 1);
        }
        seen.set(s[right], right);
        maxLen = Math.max(maxLen, right - left + 1);
    }

    return maxLen;
}
```

Critical interview gotcha: The Math.max prevents the left pointer from moving backward.

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

6. Longest Repeating Character Replacement

This problem allows an invalid window temporarily. The max frequency character is never reduced when shrinking.

```
function characterReplacement(s, k) {
    const freq = {};
    let left = 0;
    let maxFreq = 0;
    let maxLen = 0;

    for (let right = 0; right < s.length; right++) {
        freq[s[right]] = (freq[s[right]] || 0) + 1;
        maxFreq = Math.max(maxFreq, freq[s[right]]);

        while ((right - left + 1) - maxFreq > k) {
            freq[s[left]]--;
            left++;
        }

        maxLen = Math.max(maxLen, right - left + 1);
    }

    return maxLen;
}
```

Interview gold: You do NOT decrease maxFreq when shrinking. This is intentional and preserves O(n).

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

7. Remaining Sliding Window Patterns

• Permutation in String → fixed-size window + frequency map
• Minimum Window Substring → dynamic window with exact counts
• Sliding Window Maximum → deque-based window optimization

These problems reinforce that sliding window is not about window size, but about maintaining the correct invariant efficiently.