ChatGPT

Sliding Window Pattern in FAANG Interviews: A Deep Dive

Introduction

The **Sliding Window** pattern is a specialized two-pointer technique where both pointers move in the same direction to define a *contiguous window* within an array or string [1]. In a sliding window, one pointer typically marks the start of the window and the other marks the end, and together they "slide" through the data structure from left to right. This differs from the opposite-end two-pointer approach (e.g. two-sum on a sorted array or the container-with-most-water problem) where pointers start at opposite ends and move toward each other [1]. The sliding window is essentially the *cousin* of the generic two-pointer technique: we still use two pointers, but in a specific way – maintaining a window that either expands or contracts to satisfy certain conditions [2].

How to recognize sliding window problems? They involve finding an optimal subarray/substring that meets a condition (maximum, minimum, contains certain elements, etc.) and the subarray must be contiguous. Common cues include phrases like “subarray” or “substring” and objectives like “longest/shortest”, “maximum/minimum sum”, or “contains all/at most k of something”. For instance, if a problem asks for a *contiguous sequence* with a property, or to find something within a range of `k` elements, it likely calls for a sliding window [3]. This pattern often turns brute-force $O(n^2)$ searches into efficient $O(n)$ solutions by reusing previous computations as the window moves [4].

Before diving into examples, note that **sliding windows only move forward** – once the right pointer advances or the left pointer increments, we never move them back. This forward-only movement ensures each element is processed at most twice (once when entering the window, once when leaving), guaranteeing linear time complexity in most cases [5]. The key is to maintain a correct **invariant** for the window: a condition that remains true as the window slides. Each example below will explicitly state the invariant and show how pointers move while preserving it.

In the following sections, we will explore the typical structure of sliding window solutions and then walk through step-by-step examples of six classic problems: 1. **Best Time to Buy and Sell Stock** (single-pass two-pointer trick) 2. **Longest Substring Without Repeating Characters** (dynamic window for unique chars) 3. **Longest Repeating Character Replacement** (window with at most *k* replacements allowed) 4. **Permutation in String** (fixed-length window using frequency counts) 5. **Minimum Window Substring** (variable window using frequency counts) 6. **Sliding Window Maximum** (fixed-size window optimized with a deque)

For each problem, we’ll identify the invariant, meticulously trace pointer movements, and explain why the algorithm works. We’ll also highlight common pitfalls (like wrong invariants) and end each with a clean JavaScript solution, including time/space complexity.

Typical Sliding Window Structure

A sliding window algorithm follows a general pattern: 1. **Define the window and invariant:** Decide what your window represents and establish an invariant condition that the window will maintain. For example, the window might represent a subarray that satisfies a constraint (such as "no duplicates" or "sum $\geq$ S"). The invariant is a one-sentence rule capturing this – e.g., *"The current window contains all required characters"* or *"The window size minus the count of the most frequent character $\leq$ k."* 2. **Expand the window (move right pointer):** We start with both pointers at the beginning. We increment the right pointer (`r`) step by step to expand the window and include new elements. As we add an element, we update the window state (such as adding a character to a frequency map, updating a running sum, etc.). We continue expanding `r` until the window *violates* the invariant or until we fulfill a condition that triggers the next phase [6]. For many problems, this means we expand until the window *becomes valid* (e.g., contains all necessary elements) [7], though in some cases we expand until it becomes *invalid* (e.g., it has one too many of something). 3. **Contract the window (move left pointer):** Once the window satisfies the condition or becomes invalid, we then move the left pointer (`l`) to shrink the window and restore the invariant or optimize the window size. Typically, while the window is valid (or invalid, depending on the scenario), we advance `l` to drop elements that are no longer needed [6] [8]. This contraction continues until the invariant is just barely satisfied again (or the window becomes valid again), meaning we've either minimized the window or fixed the violation. 4. **Record or utilize the result:** During expansion and contraction, whenever the window meets the desired condition, we record the result (e.g., update max length, update minimum length or sum, count a valid window, etc.). For example, once a window contains all required characters, you might check if it's the smallest such window seen so far [9], or if the window length is larger than the best found, update the maximum. 5. **Continue sliding:** Repeat the expand/contract cycle until the right pointer reaches the end of the input. The left pointer will also move as needed but never exceeds the right pointer. By the end, you will have considered all possible windows in an efficient manner.

This approach leverages the fact that we can update the window state **incrementally** instead of recomputing from scratch for each new window. For example, with a running sum or a frequency map, we add the new element and remove the old one in $O(1)$ time as the window moves. The window state *always reflects the contents of the current window* – this is the core invariant. In other words, *"state always matches the current window"* [10]. By maintaining this invariant, we ensure we never double-count or skip any potential window.

Frequency maps and counters: Many sliding window problems (especially on strings) use hash maps or arrays to count characters in the window. We often keep a `need` map for required counts and a `window` map for current counts [11]. The invariant might be expressed in terms of these counts (e.g., *"window has at least the needed count of each character"* for a "contains all characters" problem). We also use a counter or some metric to know when the window satisfies the requirement (for example, how many characters have met their required frequency) [11]. This allows us to check validity in $O(1)$ time at each step instead of scanning the whole window.

Expanding vs. shrinking conditions: In some problems (like finding minimum window or when a condition becomes true only after including enough elements), we expand `r` until the window is **valid** (meets the condition), then shrink `l` to optimize. In others (like when maintaining a condition that the window must always satisfy, such as "no duplicates" or "window length – maxFreq $\leq$ k"), we expand freely and shrink whenever the invariant is broken. In either scenario, at most n expansions and n contractions occur, leading to linear time complexity [5].

Monotonic optimization: Occasionally, basic sliding window isn't enough by itself to achieve $O(n)$. A prime example is **Sliding Window Maximum** where simply moving two pointers doesn't easily give the max in each window. In such cases, a *monotonic queue* (deque) is used to maintain the window's maximum or minimum. A **monotonic queue** is a data structure that behaves like a normal queue but maintains its elements in sorted (monotonic) order [12]. By pushing and popping in a way that discards smaller elements, the front of the deque is always the window's maximum. This allows updating the max in $O(1)$ time when the window slides, keeping the overall solution linear [13]. We will see this in the Sliding Window Maximum example.

With the general idea in mind, let's walk through each problem. For each, we outline how to spot the pattern, choose the invariant, the rules for moving pointers, and a step-by-step walkthrough of an example input. We then discuss why it works, common mistakes, and present the clean JavaScript solution along with complexity analysis.

Step-by-Step Walkthroughs

Best Time to Buy and Sell Stock

Problem: Given an array of stock prices, where `prices[i]` is the price on day i, find the maximum profit achievable by buying on one day and selling on a later day. (If no profit is possible, return 0.)

Why sliding window? At first glance, this problem can be solved greedily by tracking the minimum price seen so far and the best profit. However, this approach can be interpreted as a dynamic-size sliding window where the left pointer tracks the best (lowest) buy price up to the current day, and the right pointer moves through days evaluating profits [14] [15]. Both pointers move from left to right, making it a same-direction two-pointer scenario. We maintain a window `[l, r]` that represents a potential buy-sell interval with `l` as the buy day and `r` as the current sell day.

Invariant: At any point, `l` points to the index of the lowest price seen so far from day 0 up to day `r`. In other words, `prices[l]` is the minimum price in the window, ensuring the current window yields the maximum profit ending at `r` possible so far.

Pointer movement rules: - We initialize `l = 0` and start `r = 0`. Initially, the window just contains day 0. - We then increment `r` day by day. For each new day `r`: - If the price at `r` is **lower** than the price at `l` (i.e., a new lower buy price is found), we "reset" the window by setting `l = r`. This means we start a new window at the current day because buying at any earlier day would be worse (higher price) than buying at this new low price [16]. - Otherwise (the price at `r` is higher or equal to `prices[l]`), we compute the profit if we buy at `l` and sell at `r`, i.e. `profit = prices[r] - prices[l]`. We update our maximum profit if this profit is larger than the current maximum. - Move `r` to the next day and repeat until the end.

Walkthrough (Example `prices = [7, 1, 5, 3, 6, 4]`):
- **Start:** `l = 0` (price=7), `r = 0` (price=7). Max profit = 0. Window = `[7]`.
- **Day 1:** Move `r = 1` (price=1). Now `prices[r] < prices[l]` (1 < 7), so we update `l = 1`. *(New lower buy price found on day 1.)* Window is effectively reset to just `[1]`. Max profit still 0 (no sale yet).
- **Day 2:** `r = 2` (price=5). `prices[r] = 5` is greater than `prices[l] = 1`, so a profitable sale is possible. Compute profit = 5 - 1 = 4. Update max profit = 4. Window `[l..r] = [1, 5]` represents buying

at \$1 (day 1) and selling at \$5 (day 2).
- **Day 3:** `r = 3` (price=3). `prices[r] = 3` is greater than `prices[l] = 1`, profit = 3 - 1 = 2. Max profit remains 4 (since 2 < 4). Window `[1, 5, 3]` – less profit than previous sell day, so we wouldn't actually sell at 3 because 5 was better. The invariant still holds: `l` (day 1, \$1) is still the lowest price so far.
- **Day 4:** `r = 4` (price=6). `prices[r] = 6`, profit = 6 - 1 = 5. Max profit is updated to 5. Window `[1, 5, 3, 6]` now yields a better sale at \$6 (day 4) for profit 5.
- **Day 5:** `r = 5` (price=4). `prices[r] = 4`, profit = 4 - 1 = 3. Max profit stays 5. (The best window found was buying at \$1 and selling at \$6.)

By the end, max profit = 5, achieved by buying at \$1 (day 1) and selling at \$6 (day 4). This matches the known result.

Why it works: Whenever the window would yield a negative or suboptimal profit (i.e., we encounter a price lower than our current buy price), we *move the left pointer to the right pointer*, effectively starting a new window at the lower price [16]. This is logical because if a future price can yield profit, it will yield more profit when paired with the lowest possible earlier price. Any window that includes a higher buy price when a lower one later exists is pointless to consider. By always keeping `l` at the lowest price seen so far, we ensure the profit calculation `prices[r] - prices[l]` is as high as possible for each `r`. We never miss the optimal pair because every time a new minimum appears, we reset the base of the window to that new minimum, and from that point on calculate profits relative to it [16]. In effect, every potential "valley" (drop in price) is treated as a new starting point [15], and every "peak" (rise in price) from that valley is checked for profit. This approach is exhaustive yet efficient – it checks all increasing pairs implicitly and skips clearly bad pairs. As a result, it considers all possibilities and guarantees the maximum is found [17].

Common pitfalls: A wrong way to think about this is to try maintaining a fixed window size or using two loops (checking every pair i<j), which is unnecessary and inefficient. Another mistake is not resetting the left pointer when a lower price is found – if you don't move `l` and instead just calculate profits, you might hold onto an old higher price that yields a smaller profit or even negative profit, missing the chance to start fresh at a new low. The invariant "`prices[l]` is min so far" must hold; if not updated, the profit calculation can fail for future elements. Some consider this problem purely greedy (which it is), but as shown, it can be framed in the sliding window context; just remember that the "window" here can collapse to a single day when a new low is encountered.

Key takeaways: This problem demonstrates a sliding window of **dynamic size** where the window expands each day and occasionally resets (left jumps) when beneficial. The pattern to recognize is "find max difference with order constraint (i < j)" – often solved by tracking a min-so-far. The sliding window approach here ensures linear time by only moving pointers forward and never revisiting days. The invariant approach ("track lowest buy price") is crucial. In a Google interview, you should articulate that you keep two indices, update the left index whenever a new minimum appears, and update profit on the fly [18]. No nested loops are needed, and space usage is $O(1)$.

Solution (JavaScript): We'll implement the above logic. The time complexity is **O(n)** since we do a single pass through the prices (each day is visited once), and the space complexity is **O(1)**.

```
function maxProfit(prices) {
    let l = 0;                  // left pointer for buy day (min price index)
```

```
    let maxProf = 0;
    for (let r = 0; r < prices.length; r++) {
        if (prices[r] < prices[l]) {
            // Found a new lower price, update buy day
            l = r;
        } else {
            // Calculate profit if bought at l and sell at r
            const profit = prices[r] - prices[l];
            if (profit > maxProf) {
                maxProf = profit;
            }
        }
    }
    return maxProf;
}
```

Longest Substring Without Repeating Characters

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

Why sliding window? We are dealing with substrings (contiguous sequences of characters) and a condition that must hold (all characters unique). As we extend a substring by moving a right pointer, it may violate the uniqueness condition, at which point we need to slide the left pointer to restore uniqueness. This is the quintessential sliding window scenario for strings – we maintain a window that is always a substring with no duplicates [19]. Both pointers move left-to-right through the string.

Invariant: The current window `[l, r]` contains **no duplicate characters**. Equivalently, all characters in `s[l..r]` are unique.

Pointer movement rules: - Initialize `l = 0`. Use a data structure (like a set or a hashmap) to keep track of characters in the window. - Expand `r` one character at a time (from 0 to end of string): - When adding `s[r]`, if this character is **not already in the window**, simply include it (e.g., add to set) and update the max length if needed. - If the character `s[r]` **is already present** in the current window, we have a duplicate, breaking the invariant. We then enter a contraction phase: move `l` forward (and remove characters from the window) until the duplicate character is removed and the window is unique again [19]. In practice, this often means moving `l` to one position right of the first occurrence of `s[r]` in the window. - Continue this process for each `r`. The window size will expand and contract but the invariant (no duplicates) is maintained after each adjustment. - Keep track of the maximum window size encountered that satisfied the unique condition.

Walkthrough (Example `s = "abcabcbb"`):
We'll use a set to track the current window's characters.

- **Start:** `l = 0`, `r = 0`. Window = `"a"` (just first char). Unique set = {a}. Max length = 1.
- **r = 1:** char = `"b"`. It's not in the set, so expand window to `"ab"`. Set = {a, b}. Max length = 2.
- **r = 2:** char = `"c"`. Not in set, window = `"abc"`, set = {a, b, c}. Max length = 3.

- **r = 3:** char = `"a"`. This *is* a duplicate (already in set). Window `"abca"` has two 'a's, violating the invariant. We must move `l` to remove the first 'a':
- Move `l` from 0 to 1: remove `s[0] = "a"` from set. Now window is `"bca"` (from index 1 to 3) and set = {b, c, a} actually still contains 'a'? Let's do stepwise:
- After removing `a`, set = {b, c}. Now the window from 1 to 3 is `"bca"`. We then add the new `a` (at r=3) to set, so set = {a, b, c}.
- Now window = `"bca"` is unique. `l` is now 1, `r = 3`. Window length is 3, max length stays 3 (ties with previous).
- **r = 4:** char = `"b"`. Duplicate (window "bca"+"b" would have two 'b's). So we contract from the left:
- `s[1] = "b"` is at `l`; remove it and move `l` to 2. Set becomes {a, c} after removing 'b'. Now window is `"ca"` (indices 2..3) plus the new char at 4 will be "cab".
- The duplicate 'b' is gone, we can now add `s[4] = "b"` to window. Window = `"cab"` (indices 2..4), set = {a, b, c}. Unique again. Window length = 3, max still 3.
- **r = 5:** char = `"c"`. Duplicate alert (window "cab"+"c" has two 'c's). Contract:
- Current window is "cab" (indices 2..4). The new char 'c' duplicates with `s[2] = "c"` at index 2.
- Move `l` to 3: remove `s[2] = "c"` from set. Set becomes {a, b}. Window now "ab" (3..4).
- 'c' is gone, add `s[5] = "c"`. Window = `"abc"` (indices 3..5), set = {a, b, c}. Unique. Window length = 3, max still 3.
- **r = 6:** char = `"b"`. Duplicate (window "abc"+"b"). Contract:
- Window "abc" (3..5) has 'b' at index 4, new 'b' at 6.
- Move `l` to 4: remove `s[3] = "a"` (set -> {b, c}, window "bc" from 4..5). Still duplicate 'b' in window.
- Move `l` to 5: remove `s[4] = "b"` (set -> {c}, window "c" at index 5).
- Now no 'b' in window. Add `s[6] = "b"`. Window = `"cb"` (5..6), set = {b, c}. Unique. Window length = 2, max remains 3.
- **r = 7:** char = `"b"`. Duplicate (window "cb"+"b" has two 'b's).
- Window "cb" (5..6) has 'b' at 6.
- Move `l` to 6: remove `s[5] = "c"` (set -> {b}, window "b" at index 6).
- Still duplicate 'b' (window now "b" contains one 'b', new char is also 'b' at 7).
- Move `l` to 7: remove `s[6] = "b"` (set -> {} empty, window empty now).
- Add `s[7] = "b"`. Window = `"b"` (index 7), set = {b}. Window length = 1, max remains 3.

The longest substring without repeat we found was length 3 (e.g., "abc"). Indeed, the answer for "abcabcbb" is 3.

Why it works: The algorithm ensures that at all times the window only contains unique characters [19]. When a duplicate is encountered at `r`, we increment `l` just enough to remove that duplicate (and any characters before it) from the window. Importantly, we never move `l` backward, so each character is processed at most twice (once when `r` includes it, once when `l` passes it and removes it) – linear time overall. The invariant (no duplicates in window) is restored after the while-loop that moves `l` [19], so we can safely record window lengths. A common proof by invariant is that at each position `r`, our `l` is always positioned such that `s[l..r]` is the longest substring ending at `r` with no duplicates (if `l` could be further left, we wouldn't have removed characters). This means we explore every maximal unique substring exactly once. We don't miss any candidate because whenever a duplicate appears, we systematically shorten the window from the left until it's gone, effectively considering the next possible unique substring start. This greedy removal is correct because any smaller contraction would leave a

duplicate, and any larger contraction (removing more than needed) would unnecessarily discard a valid part of the substring.

Common pitfalls: A classic mistake is not removing characters in the correct order when a duplicate is found. For example, some might try to reset the window completely when a duplicate is encountered (moving `l` directly to `r`), but that skips substrings and is incorrect. Another pitfall is forgetting to remove the *earlier occurrence* of the duplicate character. One must remove *from the left* (oldest part of the window) rather than randomly removing the duplicated character from the middle. Using a set and removing leftmost chars in a loop as above is a straightforward approach. Another common error is using a frequency map but not decrementing counts properly as `l` moves, which can lead to incorrectly thinking a duplicate still exists when it has actually been removed (or vice versa). Finally, forgetting to update the max length after adjusting the window is a minor mistake – you should update after each expansion (or contraction, depending on how you write the code). A more subtle bug: if using a map of character->index (an optimized approach), one must be careful to move `l` to `max(l, lastIndex[duplicate]+1)`; failing to take the max can move `l` backwards (which is wrong).

Key takeaways: This pattern (often known as the "sliding window for longest unique substring") appears in many variants, such as "longest substring with at most K distinct characters". The invariant of maintaining no duplicates (or a constraint on distinct count) is fundamental. The problem teaches the technique of **shrinking the window** upon violation of a condition. Whenever you see a substring/subarray problem asking for a longest or shortest segment under some constraint that can be checked locally (like "no repeats" or "sum <= X"), think sliding window. Use a set or map to maintain the window state, and adjust `l` and `r` accordingly. The two-pointer method ensures $O(n)$ time because each pointer traverses the string once. Space complexity is $O(m)$ where *m* is the size of the charset (at most 26 for lowercase letters, or say 128/256 for ASCII) – effectively linear in the alphabet, or $O(n)$ in worst-case scenario (if string length n and all characters distinct, the set grows to n) [20].

Solution (JavaScript): Using a sliding window with a set to enforce uniqueness.

```
function lengthOfLongestSubstring(s) {
    const seen = new Set();
    let l = 0;
    let maxLen = 0;
    for (let r = 0; r < s.length; r++) {
        const ch = s[r];
        // If char is duplicate, shrink from left until no duplicate
        while (seen.has(ch)) {
            seen.delete(s[l]);
            l++;
        }
        // Now it's safe to include s[r]
        seen.add(ch);
        // Update max length
        const windowLen = r - l + 1;
        if (windowLen > maxLen) {
            maxLen = windowLen;
```

```
            }
        }
        return maxLen;
    }
```

Complexity: Every character is added to and removed from the set at most once, so the loop runs in O(n) time. Membership checks and deletions in a set are $O(1)$ on average. Space complexity is $O(m)$ as discussed (which is $O(n)$ in worst case, but typically $O(1)$ if we consider the character set fixed).

Longest Repeating Character Replacement

Problem: Given a string s and an integer k, you can replace at most k characters in the string with any other characters. Find the length of the longest substring that can be obtained containing all the same character after at most k replacements.

For example, s = `"AABABBA"`, k = 1. The longest substring we can get with all identical letters (by at most one replacement) is `"AABA"` (replace the single 'B' at position 2 with 'A'), which has length 4.

Why sliding window? We are asked about a *substring* (contiguous sequence) with a certain property: that it can be made all one repeating character using at most k modifications. This naturally lends itself to a window that expands until it breaks the property and contracts as needed. Specifically, if we look at any window `[l, r]`, we can ask: "how many characters in this window would need to be replaced to make it all the same?" If that number is $\leq$ k, the window is a valid candidate. As we move `r` and `l`, we maintain this information. Both pointers move forward, making it a sliding window (same-direction two-pointer) problem.

Invariant: In our approach, we maintain that *the window* `[l, r]` *is always "valid"*, meaning it could be made all one character with $\leq$ k replacements. Another way to express this invariant: `(window_length) - (count of most frequent char in window) ≤ k`. This condition means the number of chars that are *not* the dominant char in the window is at most k (and those are the ones we'd replace). As long as this holds, the window can be all one char by replacing those minority characters [21].

Pointer movement rules: - Use a frequency map (or array of size 26 for letters) to count characters in the current window. Also track `maxFreq`, the count of the most frequent character in the window. - Initialize `l = 0`. For `r` from 0 to end of string: - Add `s[r]` to the window (update its count in the freq map). Update `maxFreq` if needed (i.e., if the freq of `s[r]` becomes the new maximum). - Now check if the window is still valid: if `(r - l + 1) - maxFreq > k`, then the window has more than k characters that are not the most frequent one, which means we cannot make this window all one char with k replacements. This violates our invariant. In that case, we need to **shrink** the window from the left: - Remove `s[l]` from the window (decrement its count in the map) and increment `l` by 1. - We don't necessarily update `maxFreq` downward here because even if the freq of the previously most frequent char drops, it doesn't affect validity to not recalc immediately – we only care that if the window was invalid, we removed one char, reducing window length by 1, which eventually will restore validity [22]. (In practice, keeping an accurate `maxFreq` at all times or using a running max that might be stale both work as long as the condition check is done correctly.) - If the window was valid (or after adjustments it is valid again), record the window length. We update the answer (max length found so far) *after* possibly shrinking, or you can do it each iteration

knowing you only shrink when invalid. A common strategy is to always update the result with `r - l + 1` when the window is valid (since we only shrink exactly when needed, the window is as large as possible when valid).

- Continue until `r` reaches the end. We never move `l` backward, ensuring linear time.

Walkthrough (Example `s = "AABABBA"`, `k = 1`):
We'll use a frequency map for characters (or simply track counts of 'A' and 'B' here). We also maintain `maxFreq` in window.

- **Start:** `l = 0`, `r = 0`. Add `A`. Counts: {A:1, B:0}. `maxFreq = 1` (A appears 1). Window length = 1, condition check: `len(1) - maxFreq(1) = 0 ≤ 1` (valid). Max window size = 1.
- **r = 1:** Add `A`. Counts: {A:2, B:0}, `maxFreq = 2`. Window "AA". Check: `2 - 2 = 0 ≤ 1` (still valid). Max length = 2.
- **r = 2:** Add `B`. Counts: {A:2, B:1}, `maxFreq = 2` (still A). Window "AAB". Check: `3 - 2 = 1 ≤ 1` (valid). We could make "AAB" all 'A's by replacing 1 'B'. Max length = 3.
- **r = 3:** Add `A`. Counts: {A:3, B:1}, `maxFreq = 3`. Window "AABA". Check: `4 - 3 = 1 ≤ 1` (valid). Max length = 4.
- **r = 4:** Add `B`. Counts: {A:3, B:2}, `maxFreq = 3` (A's still most). Window "AABAB" (indices 0-4). Check: `5 - 3 = 2 > 1`. **Invalid** now, because there are 2 chars (the B's) that are not 'A', exceeding k. We must shrink:
- Remove `s[l]` which is `A` at l=0. Now `l = 1`. Counts: {A:2, B:2}. (We removed one A.) Window is now "ABAB" (indices 1-4). Window length = 4. Recompute condition: `4 - maxFreq(???).`
- We didn't recalc `maxFreq`; it was 3, but it might now actually be 2 (since we have 2 A and 2 B). However, our check uses the old `maxFreq = 3`: `4 - 3 = 1 ≤ 1` which indicates "window is valid" according to the algorithm's check. This is a subtle point: we intentionally allow `maxFreq` to be a little stale (3 instead of the true 2) because if the window was invalid, removing one char guarantees it will be valid in at most one removal. In this case, using stale `maxFreq` still correctly avoids further shrink (we exit the while after one removal). Now we consider it valid.
- Max length remains 4 (we had a length 4 valid window before as well).
- **r = 5:** Add `B`. Now at indices 1-5 window "ABABB". Counts: {A:2, B:3}, `maxFreq` might update to 3 (for B). Window length = 5. Check: `5 - 3 = 2 > 1` invalid. Shrink:
- Remove `s[1]` which was `A`. `l = 2`. Counts: {A:1, B:3}. Window "BABB" (2-5). Length 4. Check again: `4 - 3 = 1 ≤ 1` valid. Now window "BABB" is considered valid (it has 3 B's and 1 A, we can replace that 1 A with B).
- Max length still 4 (window length 4).
- **r = 6:** Add `A`. Window indices 2-6: "BABBA". Counts: {A:2, B:3}, `maxFreq = 3`. Length = 5. Check: `5 - 3 = 2 > 1` invalid. Shrink:
- Remove `s[2]` (which was `B`). `l = 3`. Counts: {A:2, B:2}. Window "ABBA" (3-6). Length 4. Check: `4 - 2 = 2 > 1` still invalid (even after removing B, we have 2 A and 2 B, `maxFreq` might have been stale 3 or actual now 2, but either way 4-3 >1 or 4-2 >1).
- Shrink again (while still invalid):
  - Remove `s[3]` (which was `A`). `l = 4`. Counts: {A:1, B:2}. Window "BBA" (4-6). Length 3. Now `maxFreq` would be 2 (for B). Check: `3 - 2 = 1 ≤ 1` valid.
- Window "BBA" is valid (we can replace one 'A' to get "BBB"). Length 3, max length still 4.

We finish with max length = 4, which matches the expected answer. One such longest valid window was "AABA" (indices 0-3) or "BBAB" (another substring of length 4 that could be all one letter by one replacement).

Why it works: The algorithm maintains the window always in a *valid* state as defined by the invariant. We only slide `l` when the window becomes invalid (i.e., needs more than k replacements) [21] [22]. By doing so, we ensure that at any given time, we are looking at the largest window (for the current `r`) that is still valid. We never shrink the window too much – only one step at a time when needed – so we don't accidentally throw away potential length. This strategy of shrinking exactly when `(window_length - maxFreq) > k` keeps the window as large as possible while still valid [22]. Because of this, we can confidently update the max length whenever we expand `r` (since we know we haven't violated the constraint beyond k). In fact, some implementations keep track of the result simply as `r - l + 1` for the largest window seen, since `l` moves only when necessary to preserve validity. The crucial observation is that `maxFreq` can be maintained (even if a bit stale) without harming correctness: if `maxFreq` refers to a character that has exited the window, the condition might be slightly off, but we only use it to decide whether to shrink. In the worst case, we might shrink a window a tad later than absolutely necessary, but it doesn't affect the final answer [22]. This is because adding more characters after that won't extend a truly invalid window's length in the result; we always realign soon enough. By the end, we have effectively tried all possible windows where replacements $\leq$ k (implicitly, without checking each one explicitly).

Another way to reason: We are trying to find the longest window for which `window_length - maxFreq <= k`. If you imagine sliding `r` and always satisfying this by moving `l` accordingly, you are effectively finding the maximum window for each position and beyond. It turns out you can just keep expanding and only shrink when you have to – this laziness ensures you don't miss the global max.

Common pitfalls: A common wrong approach is to try to shrink the window *fully until it becomes valid again* inside a loop (i.e., removing more than one character at once). While logically that also works, it complicates reasoning and isn't necessary [23] [22]. The method described (shrinking one by one in a `while` loop as needed, or equivalently checking the condition and moving `l` by one when invalid in each iteration) is simpler and ensures the invariant. Another pitfall is not updating the frequency counts correctly when moving `l` or `r`, which can lead to incorrect `maxFreq` values and thus incorrect decisions on validity. Some might forget to update `maxFreq` altogether – one trick is that you can keep a running `maxFreq` that only ever increases (never decreases) because even if the actual frequency of that char drops, it doesn't matter; the window length difference will catch the invalidity. But if you don't understand that trick, it's safer to compute `maxFreq` on the fly or update it properly. Lastly, misunderstanding what exactly needs to be $\leq$ k can cause errors: it's not the number of distinct characters, but the number of replacements needed (which is window length minus the count of the dominating char). Forgetting this formula and using something else will break the logic.

Key takeaways: This problem introduces the pattern where the window is always kept *valid*, and we look for the longest such window. The invariant is framed in terms of a formula involving the window size and a characteristic of the window (most frequent char count). Recognizing that replacing at most k characters is a constraint that can be incorporated into a sliding window check is critical. In general, whenever you have a problem about a substring that can tolerate up to k "bad" elements (like replacements, or at most k exceptions), you can often maintain a count of the "good" elements (or most frequent element in this case) and ensure the rest $\leq$ k. The use of `maxFreq` is also instructive for performance – it shows how to avoid

recomputing expensive things within the inner loop. Time complexity is O(n) since each character is processed in or out, and space is O(m) for the frequency map (m = alphabet size, 26 for English letters) [24].

Solution (JavaScript): Using a frequency array for 'A'-'Z' (26 letters). We update and check the invariant condition, shrinking `l` when needed.

```
function characterReplacement(s, k) {
    const freq = new Array(26).fill(0);
    let l = 0;
    let maxFreq = 0;
    let maxLen = 0;
    for (let r = 0; r < s.length; r++) {
        const idx = s.charCodeAt(r) - 65; // 'A' -> 0, 'B' -> 1, etc.
        freq[idx]++;
        if (freq[idx] > maxFreq) {
            maxFreq = freq[idx];
        }
        // If window is invalid (needs more than k replacements)
        while ((r - l + 1) - maxFreq > k) {
            // shrink from left
            const leftIdx = s.charCodeAt(l) - 65;
            freq[leftIdx]--;
            l++;
            // Note: we don't update maxFreq here (optional optimization)
        }
        // Now window [l..r] is valid
        const windowLen = r - l + 1;
        if (windowLen > maxLen) {
            maxLen = windowLen;
        }
    }
    return maxLen;
}
```

Permutation in String

Problem: Given two strings `s1` and `s2`, determine if `s2` contains a substring that is a permutation of `s1`. In other words, check if any anagram of `s1` appears as a contiguous substring in `s2`. Return true if yes, otherwise false.

Example: s1 = "ab", s2 = "eidbaooo". The substring `"ba"` in s2 is a permutation of s1 ("ab"), so the answer is true.

Why sliding window? We are looking for a specific-length substring in `s2` (length equal to `s1` length) that matches a condition (having the same character counts as s1, i.e., being an anagram of s1). Checking every possible substring of the appropriate length can be optimized by sliding a window of that fixed length

across `s2`. This is a classic fixed-length sliding window problem: the window size is fixed to `len(s1)` and we slide it along `s2` updating counts. Both pointers move in lockstep once the window is filled (the right pointer expands the window, and after the initial fill, every step involves moving `l` and `r` together by one).

Invariant: We maintain a window of length `len(s1)` in `s2`, and we track the character frequency difference between this window and `s1`. The ideal goal is an invariant like: *if the window is a perfect permutation of s1, then all character counts match.* While sliding, we keep counts updated such that we can quickly check if the window matches s1's frequency. (We could say the invariant is that we always know how the current window's counts compare to s1's counts.)

A more concrete invariant condition is: *We maintain the window size at* `len(s1)` *after the initial setup, and ensure that whenever the window moves, we update the counts correctly.* When the invariant holds (counts match), we have found a valid permutation.

Pointer movement rules: - Compute frequency counts for `s1` (target counts). - Initialize a window on the first `len(s1)` characters of `s2` (if `s2` is shorter than `s1`, we can return false immediately). Compute frequency counts for this window. - Check if the two frequency maps match – if yes, return true (found a permutation). - Then slide the window by 1 step each time: for each subsequent index in `s2`: - Remove the character that goes out of the window (the char at `l`) by decrementing its count. - Add the new character that comes into the window (the char at the new `r` position) by incrementing its count. - Update `l` to `l+1` (and `r` accordingly so that window length stays constant at `len(s1)`). - After each slide, compare the window's frequency map with `s1`'s frequency map to see if they match [25] [26]. - If any match is found, return true. If we finish sliding through `s2` without a match, return false.

Walkthrough (Example: s1 = "ab", s2 = "eidbaooo"):
Target counts from s1: {a:1, b:1}. Window size = 2.

- **Initial window:** take first 2 chars of s2: `"ei"`. Counts = {e:1, i:1}. Compare to {a:1, b:1}: not equal.
- **Slide 1:** window `"ei"` -> `"id"` (move one step right). Remove 'e', add 'd'. Window now covers s2[1..2] which is "id". Counts = {i:1, d:1}. Still not equal to target.
- **Slide 2:** window `"id"` -> `"db"`. Remove 'i', add 'b'. Window = "db". Counts = {d:1, b:1}. Not equal to {a:1, b:1}.
- **Slide 3:** window `"db"` -> `"ba"`. Remove 'd', add 'a'. Window = "ba". Counts = {b:1, a:1}. This **matches** the target counts {a:1, b:1} [25] [26]. We found an anagram "ba" of "ab".
- We can return true at this point.

Indeed, s2 contains "ba", which is a permutation of s1 "ab".

Why it works: We use a *fixed-length window* of size `len(s1)` and slide it through s2, so we check every possible substring of that length in s2 exactly once. By maintaining frequency counts and updating them in $O(1)$ time for each slide, we avoid the cost of recomputing counts from scratch for each substring. The invariant is that between moves, we correctly account for the character leaving and entering the window, so the frequency map for the window is always accurate [26]. If at any point it matches the frequency map of s1 exactly, it means the current window is a permutation of s1 [25]. Because we slide through all positions, we won't miss any possible substring. The check of two frequency maps can be done in $O(m)$ where m is number of possible characters (at most 26 for lowercase English letters), which is constant relative to input

size – or we can keep a count of how many characters have the required frequency (increment a `matchCount` when a frequency matches the target's frequency exactly, etc., to do checking in O(1)). But the simpler approach is just to compare maps directly, which is fine since 26 is constant.

Common pitfalls: - Not maintaining the window size properly (if one forgets to remove the char that goes out, the window grows and the comparison breaks). - Off-by-one errors in indexing when sliding. - A subtle bug is ending the loop too late or early. For example, if s2's length is equal to s1's length, you should check the initial window and then terminate (no slides needed beyond initial). Implementations should be careful to handle `len(s2) < len(s1)` (immediate false) and to ensure the slide loop covers the last possible starting index in s2 (`len(s2) - len(s1)`). - When comparing counts, a naive approach might compare 26 entries each time. This is fine (26* n operations at most), but one can also maintain a variable that tracks how many characters currently match the target count to optimize. Forgetting to update that correctly can cause errors. - Another pitfall is using sorting or other expensive operations for each window, which would be too slow (O(n * m log m) for sorting each substring of length m). The counting method is linear and correct.

Key takeaways: This is a textbook application of fixed-length sliding window. Whenever you need to find a substring of a given length (or less than or equal to a given length) that satisfies some condition dependent on a set of counts or sum, a sliding window with a fixed size is likely the way. The invariant here is simply that we always know the frequency composition of the current length-m window. It's also a good example of using two frequency maps (or one map and comparing to target) to detect an anagram. The time complexity is O(n) where n = len(s2), because we perform a constant amount of work (count updates and at most 26 checks) for each of n positions [27]. Space complexity is O(m) for the frequency arrays (where m = size of alphabet, at most 26 letters) [27].

Solution (JavaScript): We use two arrays of length 26 for counts of s1 and current window of s2. We slide through s2.

```
function checkInclusion(s1, s2) {
    const n = s2.length, m = s1.length;
    if (m > n) return false;
    const targetCount = new Array(26).fill(0);
    const windowCount = new Array(26).fill(0);
    // populate target counts from s1
    for (let ch of s1) {
        targetCount[ch.charCodeAt(0) - 97]++;  // 'a' -> index 0
    }
    // initial window of size m from s2
    for (let i = 0; i < m; i++) {
        windowCount[s2.charCodeAt(i) - 97]++;
    }
    // function to compare two count arrays
    const matchesTarget = () => {
        for (let j = 0; j < 26; j++) {
            if (windowCount[j] !== targetCount[j]) return false;
        }
```

```
        return true;
    };
    if (matchesTarget()) return true;
    // slide the window across s2
    for (let i = m; i < n; i++) {
        const inCharIndex = s2.charCodeAt(i) - 97;
        const outCharIndex = s2.charCodeAt(i - m) - 97;
        // add new char
        windowCount[inCharIndex]++;
        // remove char that slid out
        windowCount[outCharIndex]--;
        if (matchesTarget()) {
            return true;
        }
    }
    return false;
}
```

Minimum Window Substring

Problem: Given strings `s` and `t`, find the minimum window substring of `s` that contains **all** the characters of `t` (including duplicates). If no such window exists, return an empty string. This is the classic "min window substring" problem (LeetCode 76).

Example: s = `"ADOBECODEBANC"`, t = `"ABC"`. The minimum window in s that contains 'A', 'B', and 'C' is `"BANC"` (length 4).

Why sliding window? We are asked for a substring of `s` that contains all characters of `t` – a classic scenario for a sliding window with expansion and contraction. We need to find a *contiguous segment* meeting a condition (contains all required chars) and specifically the smallest such segment. A brute-force approach would check all substrings, but a sliding window can achieve this in linear time by expanding to meet the condition and then contracting to find the minimum, then repeating. Both pointers only move forward, making it efficient. The problem inherently demands checking many possible windows; sliding window allows us to do this without resetting the counts for each new start position, instead reusing the previous computation.

Invariant: We maintain a window `[l, r]` that is either in a state of **having all required characters** (a *valid* window) or is in the process of gaining them. More specifically, we maintain: - A count of characters we **need** (from t) and a count of what we **have** in the current window. - An invariant could be: *If the window is currently valid (contains all of t), then for each character* `c` *in t, the window has at least as many* `c` *as* `t` *needs.* When the window is not valid, it means it is missing at least one required character.

During the algorithm, we alternate between expanding and contracting: - When the window is not yet valid (doesn't satisfy all chars), we expand `r` to include more characters. - Once the window becomes valid (invariant satisfied), we then contract from the left to try to shorten it while still maintaining validity (to find a smaller window) [28] .

The key invariant in a valid window is: *for all chars in t, window_count[c] >= t_count[c]*.

Pointer movement rules: 1. Build a frequency map `need` for t's characters (how many of each char are needed) [11] . Also keep a `have` map for the current window counts (initially empty). 2. Have two pointers `l = 0, r = 0`. Also keep a `formed` count of how many distinct characters from t are currently satisfied in the window (i.e., `have[c] >= need[c]` for those characters). 3. Expand `r` (move right pointer) step by step, adding `s[r]` to the window counts: - If `s[r]` is a character we care about (in `need`), increment its count in `have`. If by adding this character the window now satisfies that character's requirement (i.e., `have[c] == need[c]`), increment `formed` [29] . - Continue expanding `r` until `formed` equals the number of unique characters in t (meaning the window is *valid*, containing all required chars in required counts). 4. Once the window is valid, we *try to contract* from the left (`l`) to remove any superfluous characters and potentially find a smaller valid window: - While `formed` equals the number of required unique chars (window still valid), check if the current window is the smallest so far and record its bounds if yes. - Then try to shrink: remove `s[l]` from the window (decrement its count in `have`). If `s[l]` is a char in `need` and removing it makes `have[c]` drop below `need[c]`, then the window is no longer fully valid for that char – decrement `formed` (we lost a required character) [9] [29] . - Increase `l` by 1 (contract window from left by one). - Continue this while the window remains valid, to squeeze out as much as possible. 5. After contracting, go back to expanding `r` again (the window is no longer valid if we removed something critical, so we need to expand to find the next valid window). 6. Repeat until `r` reaches end of `s`. 7. The smallest window recorded is the answer.

Walkthrough (Example: s = "ADOBECODEBANC", t = "ABC"):
Need map from t = {A:1, B:1, C:1}. Number of required unique chars = 3.

- **Initialize:** `l=0, r=0`, formed=0. have={} empty. Best window = (inf length initially).
- **Expand r=0:** s[0]='A'. 'A' is needed. have={A:1}. Since have[A] (1) == need[A] (1), formed++ (formed=1) [29] . Not all 3 formed yet.
- **r=1:** 'D' (not in need). Ignore (just add have if we want but it's irrelevant). formed still 1.
- **r=2:** 'O' (not needed). formed still 1.
- **r=3:** 'B'. Needed. have={A:1, B:1}. have[B]==need[B], so formed=2.
- **r=4:** 'E' (not needed). formed=2.
- **r=5:** 'C'. Needed. have={A:1, B:1, C:1}. have[C]==need[C], so formed=3. Now formed == 3 (all required chars satisfied). Window [l..r] = [0..5] = "ADOBEC" is valid (contains A, B, C). Window length=6.
- **Contract from l:** We try to shrink from left since window is valid.
- s[l]='A' at l=0. If we remove 'A', have[A] goes 1->0 which is < need[A], so formed would drop to 2 (no longer satisfying A) [30] . We should stop shrinking **just before** losing validity. But our algorithm typically will remove it and then break when formed drops.
- We update best window to "ADOBEC" length 6 (first valid window found).
- Remove 'A': have[A]=0, formed=2 (lost A). Increment l to 1.
- Now window [1..5] = "DOBEC" is not valid (missing A).
- **Expand r=6:** Now continue expanding since window not valid. r=6 -> 'O' (not needed).
- **r=7:** 'D' (not needed).
- **r=8:** 'E' (not needed).
- **r=9:** 'B' (needed). have[B] was 1, now have[B]=2. But need[B]=1, and formed was already counting B as satisfied. have[B] > need[B] is fine (still satisfied, formed stays 2).
- **r=10:** 'A' (needed). have[A]=1 (0->1). Now have[A] == need[A], formed++ => formed=3. Window [l..r] = [1..10] = "DOBECODEBA" now has A,B,C again. Length = 10.

- **Contract from l:** Window is valid (formed=3). Try to shrink:
- l=1 'D' (not needed) – remove it, window still valid (formed stays 3). l=2.
- l=2 'O' – remove, still valid. l=3.
- l=3 'B' – remove, have[B] goes 2->1 (still >= need, B still satisfied), formed stays 3. l=4.
- l=4 'E' – remove, valid. l=5.
- l=5 'C' – remove, have[C] 1->0 which is < need[C], so formed-- => 2. Now invalid. We removed one too many basically.
- The last removal (C) broke validity. So the last window that was valid was [l..r] = [5..10] (just before removing C). That window is "CODEBA" of length 6. Check if it's smaller than current best: current best was 6 as well ("ADOBEC"). "CODEBA" is also 6, so tie.
- However, note: we could have updated best when we had [5..10] as valid with length 6, but it's same size as best.
- Now l=6, window [6..10] = "ODEBA" invalid (missing C).
- **Expand r=11:** 'N' (not needed).
- **r=12:** 'C' (needed). have[C]=1 (0->1), formed++ => formed=3 (since A, B from before are still satisfied). Now window [6..12] = "ODEBANC" is valid (contains A,B,C). Length = 7.
- **Contract from l:** l=6 'O' remove (not needed). l=7.
- l=7 'D' remove. l=8.
- l=8 'E' remove. l=9.
- Window now [9..12] = "BANC". Still have A=1, B=1, C=1 in window (we didn't remove those).
- l=9 'B' – if remove, B goes 1->0 < need, formed would drop to 2. So stop just before removing B.
- Current window [9..12] = "BANC" is valid and length = 4. This is smaller than best (which was 6). Update best = "BANC".
- Now remove B: have[B]=0, formed=2, l=10.
- Window [10..12] invalid.
- We have reached end (r=12 was last index). Best recorded window is "BANC".

The result is "BANC", which is correct.

Why it works: This algorithm finds all windows that cover t by expanding until coverage is achieved and then shrinking to find the tightest window for that coverage [28]. By always attempting to contract when possible, we ensure that whenever we record a valid window, it's a *minimal* window for that particular right endpoint. We explore different windows by moving `l` step by step. Importantly, `l` and `r` each only move at most n steps, so the complexity is linear despite the nested-looking structure [5]. We do not restart scanning for each potential start; we pick up where we left off. The use of `formed` (or a counter of satisfied chars) lets us efficiently know when we have a valid window without checking the entire map every time, and it only updates when a requirement goes from unsatisfied to satisfied or vice versa [29]. This way, we check validity in O(1) time per iteration. By maintaining the counts in the window (`have` map) incrementally, we preserve the invariant that these counts reflect the actual content of the current window at all times. Thus we can trust our checks and the decision to contract or expand.

Common pitfalls: - Not handling cases where no valid window exists (should return ""). - Mistiming the update of the minimum window. You should update the result *after* the window becomes valid and *before* you remove a character that breaks validity. The typical pattern is: when formed == required, in a loop, update result, then remove at l, then check if that broke formed. - Off-by-one errors in extracting the substring from indices. - Forgetting to decrement the count when moving `l` or to increment when moving `r`. The bookkeeping is a bit involved; a small mistake can invalidate the algorithm. - Another pitfall: some

attempt a different invariant like "always keep window invalid and correct it", but the more straightforward is expand till valid, then shrink till just invalid.

Key takeaways: This pattern – *expand to satisfy, contract to optimize* – is very common. It applies whenever you need the smallest subarray meeting a condition (or generally all subarrays meeting a condition). The sliding window approach efficiently finds all possible valid windows without redundant work. In problems like this, using frequency maps and a count of satisfied requirements is essential for efficiency. The time complexity is O(n + m) where n = len(s) and m = len(t), because each character of s is visited at most twice (once when `r` hits it, once when `l` passes it) [5], and checking the condition is O(1). Space complexity is O(m) for the maps (bounded by alphabet size if using letters only). This problem in particular is known to be tricky in implementation, but the sliding window strategy is the clear winner for performance.

Solution (JavaScript): We implement the described approach with maps (using JS objects or Map for clarity).

```
function minWindow(s, t) {
    if (t === "" || s === "") return "";
    // Frequency map of characters needed from t
    const need = {};
    for (let ch of t) {
        need[ch] = (need[ch] || 0) + 1;
    }
    let required = Object.keys(need).length;  // number of unique chars to
satisfy
    // Sliding window counts and pointers
    const have = {};
    let formed = 0;
    let l = 0, r = 0;
    let bestLen = Infinity, bestStart = 0;
    while (r < s.length) {
        const char = s[r];
        // expand window by including s[r]
        if (need[char] !== undefined) {
            have[char] = (have[char] || 0) + 1;
            if (have[char] === need[char]) {
                formed++;
            }
        }
        // Try contracting while window is valid
        while (formed === required) {
            // update best answer
            const windowLen = r - l + 1;
            if (windowLen < bestLen) {
                bestLen = windowLen;
                bestStart = l;
            }
```

```
            // contract from left (remove s[l])
            const leftChar = s[l];
            if (need[leftChar] !== undefined) {
                have[leftChar]--;
                if (have[leftChar] < need[leftChar]) {
                    // window lost a required char
                    formed--;
                }
            }
            l++;
        }
        r++;
    }
    return bestLen === Infinity ? "" : s.substring(bestStart, bestStart +
bestLen);
}
```

Sliding Window Maximum

Problem: Given an array `nums` and an integer `k`, there is a sliding window of size `k` moving from left to right across the array. For each position of the window, return the maximum value in the window. (LeetCode 239)

Example: `nums = [1,3,-1,-3,5,3,6,7]`, `k = 3`. The windows of size 3 and their maxima are:

```
Window position                Max
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3   5] 3  6  7      5
 1  3  -1 [-3   5   3] 6  7     5
 1  3  -1  -3 [5   3   6] 7     6
 1  3  -1  -3  5 [3   6   7]    7
```

So the output is `[3, 3, 5, 5, 6, 7]`.

Why sliding window? The problem explicitly describes a sliding window of fixed size `k` moving through the array. We need the maximum for each window position. A brute force would examine each window and find the max in O(k) time, leading to O(n*k) overall, which is too slow for large n. A sliding window approach can do better by maintaining the maximum as the window moves*, reusing computations. We use the fact that as the window slides, we drop one element (left side) and add a new one (right side), and we want to update the maximum accordingly without scanning the whole window from scratch.

Invariant/Data structure: This problem introduces a special technique: using a **monotonic deque** (double-ended queue) to maintain the window's elements in decreasing order [12]. The deque will store *indices* (or values) of elements in the current window, and it will be maintained so that: - The values

corresponding to indices in the deque are in **decreasing order** from front to back. - The front of the deque is always the index of the current maximum in the window.

With this structure, as we slide the window: - Before adding a new element `nums[r]`, we remove from the back of the deque any indices whose values are smaller than `nums[r]` (because those elements can never be the max if `nums[r]` is in the window; `nums[r]` will overshadow them) [31] . - We then add the new index `r` to the back. - We remove from the front of the deque any index that falls outside the window (i.e., if `l` is the left of window, if deque's front < l, pop front). - Now the front of the deque holds the index of the largest element in the window. We output that value as the max.

This ensures we do constant work for each element (each index is pushed and popped at most once) [13] .

Pointer movement rules: This is a fixed-size window scenario. We effectively move `r` from 0 to n-1, and ensure that `l = r - k + 1` once we have the first full window. - Start by processing the first k elements (0 to k-1) with the deque setup: - For each index i in [0, k-1]: pop smaller values from back, push i. - The max for the first window is at deque's front. - Then for each subsequent index `i` from k to n-1: - Remove from front if the front index equals `i - k` (meaning it's no longer in the new window). - Pop from back while `nums[i]` is greater than values at those indices. - Push i to back. - Record `nums[deque[0]]` as the max of the current window.

Walkthrough (Example above):
We simulate the monotonic deque for the example array with k=3:

- **Initial fill (i=0,1,2):**
- i=0: deque is empty, push 0. (deque = [0] containing value 1)
- i=1: nums[1]=3, which is > nums[0]=1, so pop 0. Push 1. (deque = [1] containing value 3)
- i=2: nums[2]=-1, which is less than nums[1]=3, so no pop. Push 2. (deque = [1, 2] with values [3, -1] in descending order)
- Now the first window [0..2] max = nums[deque[0]] = nums[1] = 3.
- **Slide window: i = 3 (window will be [1..3]):**
- Before adding i=3, check front: deque front = 1, which is index of value 3. Is it out of new window [1..3]? We compare front (1) with i-k (3-3=0). 1 > 0, so it's still in window, keep it.
- nums[3] = -3. Pop from back any smaller? The back of deque is index 2 (value -1). nums[3] = -3 is not greater than -1, so we do nothing.
- Push 3. Deque now [1,2,3] corresponds to values [3, -1, -3].
- Max = nums[deque[0]]=nums[1]=3.
- **i = 4 (window [2..4]):**
- Remove out-of-window: i-k = 1. deque front is 1, which equals 1? Actually front index =1, i-k=1, so yes, the element at index1 (value3) is moving out of window. Pop front.
- Deque now [2,3] (values [-1, -3]).
- nums[4]=5. Pop smaller from back: back index 3 has value -3, which is < 5, pop it. Now back index 2 has value -1, <5, pop it. Deque now empty.
- Push 4. Deque = [4] (value 5).
- Max = nums[4]=5.
- **i = 5 (window [3..5]):**
- Remove out-of-window: i-k = 2. Front of deque =4, which is >2, so front (index4) is in window, nothing to remove.

- nums[5]=3. Back index 4 has value5, which is >3, so no pop (since we only pop if new val is greater).
- Push 5. Deque = [4,5] (values [5,3]).
- Max = nums[4]=5.
- **i = 6 (window [4..6]):**
- Remove out-of-window: i-k = 3. Front deque=4, index4 >3, in window.
- nums[6]=6. Pop from back: back index5 val3 <6, pop 5. Back index4 val5 <6, pop 4. Deque now empty.
- Push 6. Deque=[6] (val6).
- Max = nums[6]=6.
- **i = 7 (window [5..7]):**
- Remove out-of-window: i-k = 4. Check front: deque front=6 (index6) >4, in window.
- nums[7]=7. Pop from back: back idx6 val6 <7, pop6. Deque empty.
- Push7. Deque=[7] (val7).
- Max = nums[7]=7.

Max values collected: [3,3,5,5,6,7] as expected.

Why it works: The deque method is essentially maintaining the invariant that the deque's front is the max of the current window, and the deque's contents are sorted in descending order. This works because: - We always remove outdated indices (ensuring the deque only holds indices from the current window). - We remove smaller values from the back when adding a new value, so the deque always has the largest values towards the front [31]. Any value that is smaller than the new one and is behind it in the deque can never become a max in the future while the new value remains in the window, so it's safe to drop it. - Therefore, the deque's front is always the largest of those in the window. Each element is pushed once and popped at most once, so the operations overall are O(n) [13].

This approach essentially leverages a **monotonic decreasing queue** as the data structure to maintain the window's state, rather than using two pointers explicitly to enforce an invariant. It is still a sliding window approach, enhanced with a deque to handle the max query efficiently. It's a good example of combining the two-pointer idea (managing a window's range) with an appropriate data structure for aggregate queries (max).

Common pitfalls: - Forgetting to remove indices that fall out of the window (leading to incorrect maxima). - Removing the wrong end (remember to remove from front for outdated indices, from back for smaller values). - Some might try a max-heap instead of a deque. While a heap can give max in O(log n), removing outdated elements from it (lazy removal) can make it a bit more complex and overall O(n log n). The deque method is cleaner and O(n). - Off-by-one issues when computing the first output index. Typically, we start outputting from when `r` = k-1 (the first time we have a full window). Ensure to handle that correctly.

Key takeaways: This problem highlights that sliding window problems sometimes need additional data structures (like deques or heaps) to maintain complex window properties (like max or min) efficiently. The pattern recognition here is: whenever you need to frequently query something like a max or min in a sliding range, consider a **monotonic deque**. It's a known trick that appears in various contexts. The invariant in another sense: the deque always represents the current window's values in sorted order, so the max is at front. The time complexity is O(n) since each element is processed in and out at most once [13], and space complexity is O(k) for the deque.

Solution (JavaScript): We'll implement the deque approach.

```
function maxSlidingWindow(nums, k) {
    const n = nums.length;
    if (n === 0 || k === 0) return [];
    const result = [];
    const deque = []; // will store indices, maintain decreasing nums values

    for (let i = 0; i < n; i++) {
        // Remove indices out of this window (i - k)
        if (deque.length && deque[0] === i - k) {
            deque.shift();
        }
        // Remove smaller values at the back
        while (deque.length && nums[deque[deque.length - 1]] < nums[i]) {
            deque.pop();
        }
        // Add current index
        deque.push(i);
        // If window has hit size k, record the max (at deque[0])
        if (i >= k - 1) {
            result.push(nums[deque[0]]);
        }
    }
    return result;
}
```

Conclusion: Pattern Recognition Checklist

To determine if a problem can be solved by the sliding window pattern (same-direction two pointers), ask yourself: - **Does it involve subarrays or substrings?** Sliding window is a strong candidate whenever the problem is about contiguous segments of an array or string that need to satisfy a condition [3] . - **Is there a clear condition that grows or shrinks with the window?** For example, a sum, a count of distinct elements, frequency of characters, etc., that we can update incrementally as the window moves. - **Fixed size or variable size window?** If the problem specifies a fixed length (like average of every k elements, or maximum of size k window), use a fixed-length window approach (which often involves moving `r` and `l` together after initial fill). If the problem is about finding the longest/shortest such window for a condition, use a variable window that expands and contracts as needed. - **Can we maintain an invariant and update it efficiently?** For instance, no duplicates, at most k bad elements, contains all target chars. If yes, a sliding window can maintain that invariant by moving pointers and using auxiliary data structures (like sets, maps, or deques). - **Avoid brute force:** If a brute force would check all O(n^2) subsegments, likely there's a sliding window to bring it down to O(n) [4] by reusing previous computations.

Finally, remember the two broad types of two-pointer techniques: - **Opposite-direction two pointers:** used for sorted arrays or meeting in the middle (not covered in depth here). - **Same-direction (Sliding Window):** as covered, where pointers define a window that slides through the array/string. If the problem at hand

involves scanning with two indices that never move backward (monotonic movement) and using a window of data, you're almost certainly in sliding window territory.

By mastering the sliding window pattern, you unlock efficient solutions to many array and string problems common in FAANG interviews. It's all about identifying the window and invariant, then carefully managing your two pointers to maintain that invariant. Use the examples above as templates: they showcase typical scenarios like unique characters, at-most-k replacements, anagram search, covering all requirements, and optimizing range queries with a deque. With practice, you'll quickly recognize when to apply this pattern and avoid common pitfalls, allowing you to solve these problems optimally in interviews. [3] [6]

---

[1] Short Notes on Two Pointer and Sliding Window - GeeksforGeeks
https://www.geeksforgeeks.org/dsa/short-notes-on-two-pointer-and-sliding-window-1/

[2] [6] [15] [16] [17] Repeat Code With LeetCode — Best Time To Buy And Sell Stock | by Evan SooHoo | Repeat Code With LeetCode | Medium
https://medium.com/repeat-code-with-leetcode/repeat-code-with-leetcode-best-time-to-buy-and-sell-stock-ad8e491b5357

[3] [4] Sliding Window Pattern in JavaScript: A Beginner-Friendly Guide - DEV Community
https://dev.to/biswasprasana001/sliding-window-pattern-in-javascript-a-beginner-friendly-guide-2kpo

[5] [7] [8] [9] [11] [28] [29] [30] 76. Minimum Window Substring - In-Depth Explanation
https://algo.monster/liteproblems/76

[10] sliding_window package - github.com/medunes/go-algo/patterns ...
https://pkg.go.dev/github.com/medunes/go-algo/patterns/sliding_window

[12] [13] [31] Monotonic Queue to Solve Sliding Window Problems
https://labuladong.online/en/algo/data-structure/monotonic-queue/

[14] [18] Best Time to Buy and Sell Stock - Why is it 'Sliding window; : r/leetcode
https://www.reddit.com/r/leetcode/comments/16dp0dq/best_time_to_buy_and_sell_stock_why_is_it_sliding/

[19] [20] Sliding Window Algorithm Breakthrough: Understanding Invariants | Noman Sajid posted on the topic | LinkedIn
https://www.linkedin.com/posts/noman-sajid01_dsa-javascript-slidingwindow-activity-7418698291678240768-oNW-

[21] [24] Sliding Window - Longest Repeating Character Replacement | A Developer Diary
https://adeveloperdiary.com/algorithm/sliding-window/longest-repeating-character-replacement/

[22] [23] 424. Longest Repeating Character Replacement - In-Depth Explanation
https://algo.monster/liteproblems/424

[25] [26] [27] Permutation in String
https://interviewing.io/questions/permutation-in-string