Topic Introduction

Today's PrepLetter is all about **the Sliding Window technique** — a powerful pattern that lets you efficiently process contiguous subarrays or substrings of a given size. If you've heard of it but haven't quite "clicked" with it yet, this is your moment!

What is the Sliding Window technique?

The sliding window is a neat approach that helps you avoid redundant work when handling problems that involve checking or computing something about every subarray, substring, or segment of a fixed (or sometimes variable) length.

How it works:
Imagine you have a window of size `k` sitting at the start of an array. You process whatever you need inside that window, then "slide" the window one element to the right, updating your calculations as you go. The trick is to update the result *incrementally* — only considering the new element coming in and the old one leaving — instead of recomputing everything from scratch.

When and why is it useful?
Sliding window shines when you need to solve problems like:

- Find the maximum/minimum/sum/average in every subarray of length `k`.
- Find all substrings of a given length that meet some property (like being an anagram).
- Find the length of the longest subarray meeting some condition.

Since many brute-force solutions for these problems are O(nk), using a sliding window can often cut these down to O(n).

Simple Example (not one of today's problems):
Say you want the sum of every subarray of length 3 in `[1, 2, 3, 4, 5]`.
Instead of summing each subarray from scratch, you can:

- Compute the first sum: 1 + 2 + 3 = 6
- For each step, subtract the element that's leaving and add the new one:
    - Next sum: 6 - 1 + 4 = 9
    - Next sum: 9 - 2 + 5 = 12

This idea — updating your result as you slide — is the heart of the sliding window.

Meet Today's Problems

The sliding window isn't just for sums! Today, we'll use it in three classic ways:

- **Sliding Window Maximum** (LeetCode 239)
- **Sliding Window Minimum** (LeetCode 239 variant, and [Minimum])
- **Find All Anagrams in a String** (LeetCode 438)

Why these three?
All three demand efficient handling of overlapping windows, but each puts its own spin on things. The first two use a *deque* to efficiently track the max or min in each window. The third uses the sliding window idea with a *frequency count* to find all substrings

that are anagrams of a given string.

Let’s dive in, starting with the classic and slightly trickiest: finding the maximum in every sliding window.

Problem 1: Sliding Window Maximum

Problem Statement (Rephrased):
Given an array of integers and a window size `k`, return a list of the maximum value in every contiguous window of size `k` as the window slides from left to right.

LeetCode 239: Sliding Window Maximum

Example:
Input: `nums = [1, 3, -1, -3, 5, 3, 6, 7]`, `k = 3`
Output: `[3, 3, 5, 5, 6, 7]`

How?

- The first window is `[1, 3, -1]` → max is 3
- Next window `[3, -1, -3]` → max is 3
- Next window `[-1, -3, 5]` → max is 5
- ... and so on.

Your turn:
Try this input manually:
`nums = [9, 11, 8, 5, 7, 10]`, `k = 2`
What should the output be?

Brute-force approach

Loop through every window, scan each window to find its max.

- Time: O(nk) (bad if n is big and k is not tiny)

Optimal approach: Deque-based Sliding Window

Here’s the trick:

- Use a double-ended queue (deque) to store *indices* of useful elements.
- The deque always stores indices of elements in **decreasing order** (from front to back).
- The front of the deque is always the index of the max for the current window.
- As you slide:
    - Remove indices that are out of the window.
    - Remove indices from the back whose corresponding values are less than the new value (they’ll never be max again!).

Step-by-step:

- Loop through the array.
- For each element:

- Pop indices from the back if their value is less than or equal to current element.
- Add current index to the back.
- Pop front if it's outside the window.
- If you've processed at least k elements, append the value at the front index to the output.

Let's code this up!

```
from collections import deque

def maxSlidingWindow(nums, k):
    """
    Returns a list of the maximums of every contiguous subarray of size k.
    Args:
    nums: List[int] - the input array
    k: int - window size

    Returns:
    List[int] - list of window maximums
    """
    if not nums or k == 0:
        return []

    dq = deque()  # Will store indices of useful elements
    result = []

    for i, num in enumerate(nums):
        # Remove indices of elements not in the current window
        while dq and dq[0] <= i - k:
            dq.popleft()

        # Remove from back while current num is bigger
        while dq and nums[dq[-1]] <= num:
            dq.pop()

        dq.append(i)

        # Append max (front of deque) to result, starting from i >= k - 1
        if i >= k - 1:
            result.append(nums[dq[0]])

    return result
```

Time and Space Complexity

- **Time:** O(n). Every index is added and removed from the deque at most once.
- **Space:** O(k) for the deque and O(n - k + 1) for the result.

Code Walkthrough

- `dq` holds indices in decreasing order of their corresponding values.
- For each index `i`:
    - Remove elements from the back if they are less than or equal to `nums[i]` (since `nums[i]` is now the candidate for max).
    - Remove the front if it's out of the window (`i - k`).
    - The max for the current window is always at the front.
- Start adding results only after reaching at least the first complete window.

Trace Example

Let's trace `nums = [1, 3, -1, -3, 5, 3, 6, 7]`, `k = 3`:

- i=0, num=1: dq=[0]
- i=1, num=3: pop 0 (since 3 >= 1), dq=[1]
- i=2, num=-1: dq=[1,2] (since -1 < 3)
    - i >= 2: result.append(nums[1]) -> 3
- i=3, num=-3: dq=[1,2,3]
    - i >= 2: result.append(nums[1]) -> 3
- i=4, num=5: pop 3, pop 2, pop 1 (since 5 >= all those), dq=[4]
    - i >= 2: result.append(nums[4]) -> 5
- ...and so on.

Try this yourself:
Input: `[2, 1, 3, 4, 6, 3, 8, 9, 10, 12, 56]`, `k = 4`
What should the output be?

Reflect:
This problem can also be solved using a heap or a balanced BST, but with more overhead. The deque method is optimal and elegantly demonstrates the sliding window!

Problem 2: Sliding Window Minimum

Problem Statement (Rephrased):
Given an array of integers and a window size `k`, find the minimum value in every contiguous window of size `k` as the window slides across the array.

Sliding Window Minimum (LeetCode 239 variant)

Example:

Input: `nums = [1, 3, -1, -3, 5, 3, 6, 7]`, `k = 3`

Output: `[ -1, -3, -3, -3, 3, 3 ]`

Your turn:

Try this input:

`nums = [4, 2, 12, 3, 1, 6, 8]`, `k = 3`

What's the output?

Brute-force

Again, for each window, scan for the min.

- Time: O(nk)

Optimal Approach: Deque for Minimum

This is nearly identical to the maximum version, but:

- Store indices in **increasing order** of their values.
- Remove from the back while the current number is **less than or equal** to the values at those indices.

Step-by-step:

- Loop through the array.
- For each element:
    - Pop indices from the back if their value is **greater than or equal** to the current element.
    - Add the current index to the back.
    - Remove the front if it’s outside the window.
    - Once you've processed at least `k`, add the value at the front index to the result.

Pseudocode:

```
Initialize empty deque dq and result list res

For each index i in nums:
    While dq is not empty and nums[dq[-1]] >= nums[i]:
        Remove dq[-1]
    Add i to dq
    If dq[0] <= i - k:
        Remove dq[0]
    If i >= k - 1:
        res.append(nums[dq[0]])

Return res
```

Example Walkthrough

For `nums = [1, 3, -1, -3, 5, 3, 6, 7]`, `k = 3`:

- Window 1: [1,3,-1] → -1
- Window 2: [3,-1,-3] → -3
- Window 3: [-1,-3,5] → -3
- Window 4: [-3,5,3] → -3
- Window 5: [5,3,6] → 3
- Window 6: [3,6,7] → 3

Try this manually:

Input: `[10, 8, 7, 5, 10, 6]`, `k = 2`

Complexity

- **Time:** O(n) — each index is added and removed at most once.
- **Space:** O(k) for deque, O(n - k + 1) for output.

Problem 3: Find All Anagrams in a String

Problem Statement (Rephrased):

Given a string `s` and a pattern `p`, find all starting indices in `s` where a substring is an anagram of `p`.

LeetCode 438: Find All Anagrams in a String

Example:

Input: `s = "cbaebabacd"`, `p = "abc"`

Output: `[0, 6]`

(Substrings starting at index 0 ("cba") and 6 ("bac") are anagrams of "abc")

Try this input:

`s = "abab"`, `p = "ab"`

What should the output be?

Brute-force

For every substring of length len(p), check if it's an anagram (by sorting or counting).

- Time: O(n *k) or worse (with sort: O(n* k log k))

Optimal Approach: Sliding Window with Frequency Count

Here we use a sliding window of length `len(p)`, and compare character counts between the window and `p`.

Step-by-step:

- Build a frequency map of `p`.
- Initialize a window frequency map for the first `len(p)` chars of `s`.

- As you slide the window:
    - If the two maps are equal, record the starting index.
    - Remove the count of the char leaving the window.
    - Add the count of the char entering.
- Repeat until you reach the end.

Pseudocode:

```
If len(s) < len(p): return []

Build count_p: frequency map of p
Build count_window: frequency map of first len(p) chars of s
Initialize result list

For i from 0 to len(s) - len(p):
    If count_window == count_p:
        result.append(i)
    Remove count of s[i] from count_window (leftmost char)
    Add count of s[i + len(p)] to count_window (next char, if exists)

Return result
```

Example Walkthrough

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

- First window: "cba" → matches "abc" → index 0
- Slide to "bae" → doesn't match
- Slide to "aeb" → doesn't match
- ...
- At index 6: "bac" → matches "abc" → index 6

Try this:

`s = "afdgacbda"`, `p = "abc"`

Which indices are anagrams?

Complexity

- **Time:** O(n), where n is the length of `s` (since count comparisons and updates can be done in O(1) for small alphabets).
- **Space:** O(1) (since the alphabet is fixed, e.g., lowercase English letters).

Summary and Next Steps

You've just unlocked a key interview pattern: **Sliding Window**.

• Both maximum and minimum sliding window problems use a deque to maintain efficient access to max/min in O(1) per window.

• Anagram-finding uses sliding window with frequency counting for substring matching.

• These patterns turn brute-force O(nk) solutions into sleek O(n) ones — a *huge* difference for big inputs!

Common mistakes:

• Forgetting to remove outdated indices/elements from your window.

• Not updating window size correctly when adding/removing elements.

• Comparing maps incorrectly in the anagram problem (e.g., not handling 0 counts).

Keep these variations in mind:

• Sometimes, the window size changes dynamically (e.g., "longest substring with at most k distinct chars").

• For min/max, heaps work but are usually less efficient than deques.

Action Steps

• **Solve all 3 problems on your own** — even the one with code here. Dry-run and hand-trace the deque!

• Try solving Problem 2 and 3 using a different technique, like heaps or brute-force, to see the trade-offs.

• Explore related problems: "Longest Substring Without Repeating Characters," "Minimum Window Substring," "Subarray Sum Equals K."

• Compare your solution with others: focus on edge cases and code clarity.

• If you get stuck, review the pseudocode and trace small examples — practice is how the pattern sticks!

Remember:

Sliding window problems are everywhere. The more you practice, the more "aha!" moments you'll have. Happy coding!