Topic Introduction

Today we're diving into the world of **Sliding Window** algorithms! This is a go-to strategy for many substring and subarray problems, especially when you want to efficiently examine all possible segments of a sequence without redundant work.

What is Sliding Window?
A "window" is simply a range (often defined by two pointers) over your array or string. Instead of looking at every possible start and end index (which can be slow), the sliding window technique moves these pointers in a smart way to consider one section at a time, updating results as it grows, shrinks, or slides.

How does it work?

- Start with both pointers (let's call them `left` and `right`) at the beginning.
- Expand the `right` pointer to include more elements in the window.
- When the window violates a constraint (like repeating characters or too many distinct characters), move the `left` pointer forward to shrink the window until the constraint is satisfied again.
- At every step, you can check the length or the property of your current window.

Why is it useful in interviews?
Many substring or subarray problems ask for the "longest" or "shortest" range with certain properties. The sliding window shines here because it lets you process every character only a few times, yielding O(n) solutions where brute-force would be O(n^2).

Simple Example
Suppose you want to find the maximum sum of any subarray of length 3 in `[2, 1, 5, 1, 3, 2]`.

- Use a window of size 3, add up the first three elements.
- Slide the window one step to the right: subtract the element going out (`2`), add the new element coming in (`1`), and repeat.
- No need to repeatedly sum every possible subarray!

Why These Three Problems Together?

Today's trio are all classic *substring* problems that ask for the **longest possible substring** under a certain constraint:

- No repeating characters.
- At most K distinct characters.
- At most K replacements to make all characters the same.

Each one is a perfect showcase for the sliding window: you need to efficiently expand and contract your window to maintain the constraint and track the best (longest) result as you go.

Problem 1: Longest Substring Without Repeating Characters

LeetCode 3

Problem Restatement
Given a string, find the length of the longest substring that contains no repeating characters.

Example

Input: `"abcabcbb"`

Output: `3`

Explanation: The answer is `"abc"`, which has length 3. Other substrings like `"bca"` or `"cab"` also qualify.

Let's Walk Through It

Imagine you're reading the string from left to right, keeping a window of characters. Every time you see a repeated character, you need to adjust the window so that no repeats remain.

Try this test case on paper:

Input: `"pwwkew"`

Expected Output: `3` (for substring `"wke"`)

Brute-force?

Generate all substrings and check if each has all unique characters. That's O(n^3):

- O(n^2) for all substrings.
- O(n) to check if all characters are unique.

Optimal Approach: Sliding Window

- Use a set (or a dictionary) to keep track of characters in the current window.
- Expand `right` pointer to add new characters.
- If you see a repeat, move `left` forward and remove characters until the repeat is gone.
- Keep track of the maximum window size seen.

Here's the code in Python:

```
def lengthOfLongestSubstring(s):
    # Set to store unique characters in the current window
    seen_chars = set()
    left = 0
    max_len = 0

    # Iterate with the right pointer
    for right in range(len(s)):
        # If s[right] is a duplicate, move left pointer forward
        while s[right] in seen_chars:
            seen_chars.remove(s[left])
            left += 1
        seen_chars.add(s[right])
        max_len = max(max_len, right - left + 1)
    return max_len
```

Time complexity: O(n), since each character is added and removed at most once.

Space complexity: O(min(n, m)), where m is the size of the character set.

Code Walkthrough

- `seen_chars` tracks what's in the current window.
- `left` and `right` form your sliding window.
- For each `right`, if the character is already in the window, move `left` forward, removing characters until no duplicates remain.
- Update `max_len` whenever you extend the window.

Trace Example

Let's trace `"pwwkew"`:

Step	left	right	seen_chars	Current char	Action	max_len
0	0	0	{}	'p'	Add 'p'	1
1	0	1	{'p'}	'w'	Add 'w'	2
2	0	2	{'p','w'}	'w'	Remove 'p',	
	1	2	{'w'}		Remove 'w',	
	2	2	{}		Add 'w'	2
3	2	3	{'w'}	'k'	Add 'k'	2
4	2	4	{'w','k'}	'e'	Add 'e'	3
5	2	5	{'w','k','e'}	'w'	Remove 'w',	
	3	5	{'k','e'}		Add 'w'	3

Try this test case yourself:

Input: `"bbbbb"`

What's the answer? (Expected: `1`)

Take a moment to solve this on your own before jumping into the code above!

Reflective prompt:

Did you know you could also solve this using a dictionary to store the last seen index of each character? Try implementing that version after you finish!

Problem 2: Longest Substring with At Most K Distinct Characters

LeetCode 340

Problem Restatement

Given a string `s` and an integer `k`, find the length of the longest substring that contains at most `k` distinct characters.

Example

Input: `s = "eceba"`, `k = 2`

Output: `3`

Explanation: `"ece"` is the longest substring with at most 2 distinct characters.

How is this Different?

Instead of no repeats, the constraint is now the *number* of distinct characters. You need to track counts of each character, not just

their presence.

Try this test case manually:

Input: `"aa"`, `k = 1`

Expected Output: `2`

Brute-force?

Try every substring, count distinct characters. O(n^3): O(n^2) substrings, O(n) to count unique chars.

Optimal Approach: Sliding Window + Hash Map

- Use a hashmap to count occurrences of each character in the current window.
- Expand `right` pointer and add chars.
- If you exceed `k` distinct chars, move `left` pointer forward and decrement counts, removing from map when count hits zero.
- Track the maximum window length.

Pseudocode:

```
function lengthOfLongestSubstringKDistinct(s, k):
    if k == 0: return 0
    char_count = empty map
    left = 0
    max_len = 0

    for right in 0 to len(s) - 1:
        add s[right] to char_count (increment count)
        while number of keys in char_count > k:
            decrement count of s[left] in char_count
            if count of s[left] == 0:
                remove s[left] from char_count
            left += 1
        max_len = max(max_len, right - left + 1)
    return max_len
```

Example Trace

Let's trace `"eceba"`, `k=2`:

- `right=0`: 'e', map: {'e':1}
- `right=1`: 'c', map: {'e':1,'c':1} (window valid)
- `right=2`: 'e', map: {'e':2,'c':1} (window valid)
- `right=3`: 'b', map: {'e':2,'c':1,'b':1} (3 distinct, shrink)
    - Remove from left: 'e' count->1
    - Remove from left: 'c' count->0, remove 'c'
    - Now map: {'e':1,'b':1}
- Continue...
- Longest valid window: `"ece"` (length 3)

Try this test case:

Input: `"abcadcacacaca"`, `k=3`

What's the answer? (Try working it out!)

Time complexity: O(n) - each character enters and leaves the window at most once.

Space complexity: O(k) for the hashmap.

Problem 3: Longest Repeating Character Replacement

LeetCode 424

Problem Restatement

Given a string `s` and integer `k`, return the length of the longest substring you can get by replacing at most `k` characters to make all characters in the window the same.

Example

Input: `s = "AABABBA"`, `k = 1`

Output: `4`

Explanation: Replace one 'A' to get "AABBBBA" or "AAABBAA", so the answer is 4.

What's New Here?

Now, instead of a strict limit on distinct characters, you can make up to `k` changes in the window to maximize the length of a substring with all the same character.

Key Insight

The smallest number of changes needed to make the window all the same character is:

`window size - count of the most frequent character in the window`

If that is <= k, the window is valid.

Pseudocode:

```
function characterReplacement(s, k):
    count = empty map
    max_count = 0  // max freq of any single char in window
    left = 0
    result = 0

    for right in 0 to len(s) - 1:
        increment count[s[right]]
        update max_count to max(max_count, count[s[right]])
        while (right - left + 1) - max_count > k:
            decrement count[s[left]]
            left += 1
        result = max(result, right - left + 1)
    return result
```

Example Trace

For `"AABABBA"`, `k=1`:

- Expand right, updating counts and `max_count`.
- When window size minus `max_count` exceeds `k`, move left pointer forward.
- Maximum window found is length 4.

Try this test case:
Input: `"ABAB"`, `k=2`
What's the answer? (Expected: 4)

Time complexity: O(n), each index processed at most twice.
Space complexity: O(1), since only 26 uppercase letters.

Reflective prompt:
Notice how this sliding window variation requires maintaining the max frequency, and the shrinking condition is different from the previous problems. Can you think of other problems where you need to maintain a "most frequent" statistic in a window?

Summary and Next Steps

These three problems are grouped because they're all about **optimizing substrings with a sliding window**, but each one tweaks the constraint:

- No repeats.
- At most K distinct.
- At most K changes.

The sliding window pattern lets you process these in O(n) time by expanding and contracting a window and tracking properties (unique characters, counts, frequencies).

Key Patterns to Remember:

- Use a set for unique items, a hashmap for counts, or a variable for max frequency.
- Adjust the shrinking condition based on the problem's constraint.
- Always update your result after expanding or shrinking the window.

Common Mistakes:

- Forgetting to update counts or remove from map when count hits zero.
- Not updating the result at the right time.
- Off-by-one errors when calculating window size.

Action List

- Solve all three problems on your own—even the one with code provided.
- Try writing Problem 2 and 3 using a different approach (such as a frequency array instead of a dictionary).
- Seek out other sliding window substring problems (like "Minimum Window Substring" or "Permutation in String").
- Compare your solutions to others—especially for tricky edge cases.
- If you get stuck, review the step-by-step logic and keep practicing. Mastery comes from repetition!

Happy coding, and may your windows always slide smoothly!