Topic Introduction

Welcome back, coding champion! Today, we're diving deep into the **sliding window** technique. If you've ever felt lost in a sea of substring problems, this PrepLetter is for you.

What is Sliding Window?
Sliding window is a strategy for efficiently handling problems involving subarrays or substrings in a larger array or string. The core idea is to maintain a window (a range defined by two pointers) that slides across your input, expanding or shrinking as needed to satisfy certain conditions.

How does it work?
Imagine you're looking for something (like the longest substring with unique characters) without having to check every possible substring. Instead of repeatedly starting from scratch, you "slide" your window: increase the right pointer to expand, and move the left pointer forward to shrink, always tracking the best answer along the way.

Why is it useful in interviews?
Many string and array problems ask for "shortest/longest subarray/substring with property X." The brute-force way is slow, but sliding window often solves them in linear time. Showing this skill in an interview wins big points for both speed and insight!

Quick Example (not one of today's problems):
Suppose you want the longest substring with at most 2 distinct characters in "eceba".
With sliding window, you expand the right end as long as you have at most 2 unique characters, and move the left end forward when you get more.
"ece" and "ceba" both have at most 2 unique letters; the answer is length 3.

Why These Problems are Grouped Together

Today's trio are all classic **sliding window** problems that focus on substrings and optimizing for length or content:

- **Minimum Window Substring:** Find the smallest substring that contains all the characters from another string.
- **Longest Substring Without Repeating Characters:** Find the longest substring with all unique characters.
- **Longest Repeating Character Replacement:** Find the longest substring you can get by replacing at most k characters to make all characters the same.

All three require careful management of window boundaries and character counts. Mastering them means you'll recognize and crack similar problems with confidence!

Problem 1: Longest Substring Without Repeating Characters

LeetCode #3

Problem Restated:
Given a string `s`, find the length of the longest substring that contains no repeating characters.

Example Input/Output:

Input: `s = "abcabcbb"`

Output: `3`

Explanation: The answer is `"abc"`, which has length 3.

How to Think About It:

This is the classic "find the longest substring with a property" problem. Since we care about unique characters, every time we see a repeat, our window must shrink until the repeated character is gone.

Pen-and-paper encouragement:

Try "pencil tracing" the solution on paper for "pwwkew" — it builds deep intuition!

Extra Test Case:

Input: `s = "bbbbb"`

Expected Output: `1`

Brute-Force Approach

Check every substring, and for each, check if all characters are unique.

- For string of length n, there are O(n^2) substrings, and each substring could take O(n) to check uniqueness.
- **Total time:** O(n^3) — far too slow!

Optimal Approach: Sliding Window

Core Pattern:

- Use two pointers (left, right) to define the window.
- Use a set to keep track of unique characters in the window.
- Expand right pointer; if a repeat is found, move left pointer forward and remove from set until window is unique again.
- Track the maximum length found.

Step-by-step Logic

- Initialize a set for seen characters, `left` pointer at 0, and `max_length` at 0.
- Iterate `right` from 0 to n-1:
  - If `s[right]` not in set, add it and update `max_length`.
  - If `s[right]` is in set, remove `s[left]` from set and advance `left` until `s[right]` can be added.
- Return `max_length`.

Python Solution

```
def lengthOfLongestSubstring(s):
    seen = set()                # Stores unique characters in window
    left = 0
    max_length = 0
```

```
    for right in range(len(s)):
        while s[right] in seen:
            seen.remove(s[left])  # Remove from window left side
            left += 1             # Move left pointer forward
        seen.add(s[right])        # Add current char to window
        max_length = max(max_length, right - left + 1)
    return max_length
```

Time Complexity: O(n) — Each character is added and removed at most once.
Space Complexity: O(min(n, m)) — m is the charset size.

Code Explanation

- `seen` keeps track of the current window's characters.
- `left` and `right` are window boundaries.
- If a duplicate is found, shrink the window from the left until it's gone.
- Update `max_length` each time the window grows.

Example Trace

Let's trace `s = "abcabcbb"`:

- right=0, s[0]='a': add 'a', window="a", max=1
- right=1, s[1]='b': add 'b', window="ab", max=2
- right=2, s[2]='c': add 'c', window="abc", max=3
- right=3, s[3]='a': 'a' is duplicate; remove 'a' at left=0, left=1, add 'a'
  - window="bca"
- Continue, always updating max_length to stay at 3

Try this test case yourself:
`s = "dvdf"`
Expected Output: `3` (from "vdf" or "dvd")

Take a moment to try solving this on your own before checking the code!

Problem 2: Minimum Window Substring

LeetCode #76

How it's similar:
Still sliding window, but now you must find the **smallest substring** in `s` that contains **all characters** of string `t` (including duplicates).

Problem Restated:
Given two strings `s` and `t`, return the minimum window in `s` which contains all the characters in `t`. If no such window exists, return

the empty string.

Example:

Input: `s = "ADOBECODEBANC", t = "ABC"`

Output: `"BANC"`

Explanation: The smallest substring of `s` containing 'A', 'B', and 'C' is "BANC".

Try this test case:

`s = "a", t = "aa"`

Expected Output: `""` (since there aren't enough 'a's in `s`)

Brute-Force Approach

Check all substrings of `s` and, for each, check if it contains all characters of `t`.

- Too slow: O(n^3) — Not practical for interview.

Optimal Approach: Sliding Window with Character Counts

Key Differences from Problem 1:

- Need to match counts of each character (not just unique presence)
- Use a hash map to track required characters and counts

Step-by-step Logic

- Build a dictionary `need` to count each character in `t`.
- Use two pointers (`left`, `right`) to define the sliding window.
- Use a second dictionary `window` to count characters in the current window.
- Expand `right` to include more characters.
- When window contains all needed characters (using a `formed` variable), try to shrink from the left.
- Keep track of the smallest window that works.

Pseudocode

```
need = counts of t's characters
window = empty counts
left = 0
min_length = infinity
min_window = ""

for right in 0..len(s)-1:
    add s[right] to window
    if window meets all needs:
        while window meets all needs:
```

```
            update min_length, min_window if smaller window found
            remove s[left] from window
            left += 1

return min_window
```

Time Complexity: O(n)

Space Complexity: O(m + n) (size of s and t's alphabet)

Example Trace

`s = "ADOBECODEBANC", t = "ABC"`

- Expand window to cover all of 'A', 'B', 'C'
- Shrink from left to find the smallest such window
- At the end, "BANC" is the smallest valid window

Dry-run this input:

`s = "aaflslflsldkalskaaa", t = "aaa"`

Expected Output: `"aaa"`

Problem 3: Longest Repeating Character Replacement

LeetCode #424

What's different or more challenging:

Now, you're allowed to replace up to `k` characters in the substring to maximize the length of a substring where all characters are the same.

Problem Restated:

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

Example:

`s = "AABABBA", k = 1`

Output: `4`

Explanation: Replace one 'A' with 'B' or one 'B' with 'A' to get "AABA" or "BBBB".

How is it similar?

Still a sliding window, but now the window can tolerate up to `k` "bad" characters.

Brute-Force Approach

Try every substring, count the most frequent character, and see if the rest can be replaced in k steps.

- O(n^3) — not efficient!

Optimal Approach: Sliding Window with Frequency Map

Core Insight:

- For each window, only need to know the most frequent character's count.
- If (window size - max_freq) <= k, window is valid.

Pseudocode

```
left = 0
max_count = 0
char_counts = empty map
result = 0

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

Test Case to Try:

`s = "ABAB", k = 2`

Expected Output: `4` (replace both 'A's or both 'B's)

Time Complexity: O(n)

Space Complexity: O(1) (since character set is small, e.g., uppercase English letters)

Reflect:

Can you see how this pattern relates to the previous two? What if the alphabet were much larger or unbounded — how would that affect your implementation?

Summary and Next Steps

You've just explored three classic string sliding window problems:

- **Longest Substring Without Repeating Characters:** Expand and shrink to maintain unique characters.
- **Minimum Window Substring:** Expand to include all needs, shrink to minimize.
- **Longest Repeating Character Replacement:** Expand as long as you can fix the window with k changes.

Key Patterns and Insights:

- Two-pointer (left/right) window management is the heart of sliding window.
- Use sets or dictionaries to track the content or counts inside the window.

- Shrinking the window at the right moments is just as important as expanding.
- Always think: "When does my window break the rule?" and "How do I fix it?"

Common Mistakes:

- Forgetting to update the window correctly when shrinking.
- Missing edge cases (empty strings, all unique, all duplicates).
- Using unnecessary extra space (e.g., storing substrings instead of just counts).

Action List

- **Solve all three problems on your own**, even the one with code above.
- For Problems 2 and 3, try writing the solution in full code.
- See if you can implement Problem 2 using a different data structure (like collections.Counter).
- Explore similar sliding window problems — for example, "Longest Substring with At Most K Distinct Characters."
- Compare your code with others for edge-case handling and clarity.
- If you get stuck, break the problem into smaller steps and use pen and paper.

Keep practicing! The more you slide those windows, the more clearly you'll see through any substring challenge. See you in the next PrepLetter!