LeetCode 3. Longest Substring Without Repeating Characters

1. Problem Title & Link

- **3. Longest Substring Without Repeating Characters**
- https://leetcode.com/problems/longest-substring-without-repeating-characters/

2. Problem Statement (Short Summary)

Given a string s, find the length of the **longest substring** without repeating characters.

3. Examples (Input → Output)

Input: s = "abcabcbb"

Output: 3

Explanation: The answer is "abc", with length = 3.

Input: s = "bbbbb"

Output: 1

Explanation: The answer is "b", with length = 1.

Input: s = "pwwkew"

Output: 3

Explanation: The answer is "wke", with length = 3.

4. Constraints

- 0 <= s.length <= $5 * 10^4$
- s consists of English letters, digits, symbols, and spaces.

5. Thought Process (Step by Step)

This is a **variable-length sliding window** problem 💡

We maintain a window that always contains *unique characters*.

We use a **set (or map)** to track which characters are currently inside the window.

Step 1: Initialize Window

Use two pointers:

- left → window start
- right → window end

Expand right to include new characters until a repeat occurs.
When a repeat is found, **shrink the window** from the left until the duplicate is removed.

Step 2: Track Max Length

At each step, update max_len = max(max_len, right - left + 1)

6. Pseudocode

set = empty
left = 0
max_len = 0

for right in range(0, len(s)):
  while s[right] in set:
    remove s[left]
    left += 1
  add s[right] to set
  max_len = max(max_len, right - left + 1)

return max_len

7. Code Implementation

Python

```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        seen = set()
        left = 0
        max_len = 0

        for right in range(len(s)):
            while s[right] in seen:
                seen.remove(s[left])
                left += 1
            seen.add(s[right])
            max_len = max(max_len, right - left + 1)

        return max_len
```

Java

```
class Solution {
    public int lengthOfLongestSubstring(String s) {
        Set<Character> set = new HashSet<>();
        int left = 0, maxLen = 0;

        for (int right = 0; right < s.length(); right++) {
            while (set.contains(s.charAt(right))) {
                set.remove(s.charAt(left));
                left++;
            }
            set.add(s.charAt(right));
            maxLen = Math.max(maxLen, right - left + 1);
        }
        return maxLen;
    }
}
```

8. Time & Space Complexity

- **Time:** O(n) — each character is visited at most twice.
- **Space:** O(k) — k distinct characters in current window.

9. Dry Run (Step-by-Step Execution)

👉 Input: s = "abcabcbb"

Step	left	right	Window	Action	max_len
1	0	0	"a"	add 'a'	1
2	0	1	"ab"	add 'b'	2
3	0	2	"abc"	add 'c'	3
4	0	3	"abca"	a' repeats → remove until left=1	3
5	1	3	"bca"	unique	3
6	1	4	"bcab"	repeat 'b' → remove until left=2	3
7	2	5	"cab"	unique	3

Output: 3 (substring “abc”)

10. Concept Insight Table

Core Concept	Common Use Cases	Common Traps	Builds / Next Steps
Sliding Window + HashSet (Dynamic Shrink) — maintain a unique window and expand/contract on duplicates.	- Unique substring problems - Longest/ shortest window tasks - Real-time stream distinct tracking	- Forgetting to remove left chars on duplicate - Off-by-one in window size - Using wrong data structure (needs O(1) lookup)	🔷 Builds to **LeetCode 76 (Minimum Window Substring)** 🔷 Related to **LeetCode 159 (Longest Substring with 2 Distinct Chars)** 🔷 Reinforces pattern for string & array window problems

11. Common Mistakes / Edge Cases

- Forgetting to move left correctly when duplicates occur.
- Counting substring length incorrectly (right - left + 1).
- Misusing set vs map (set is fine here since we only care about existence).

12. Variations / Follow-Ups

- **LC 159** → Longest substring with at most 2 distinct characters.
- **LC 340** → Generalized to *at most k distinct*.
- **LC 76** → Minimum window that *contains* all characters (reverse logic).