Longest Substring Without Repeating Characters — Illustrated Guide

A Light■Theme PDF explaining the sliding■window approach, visuals, table trace, and code for the classic interview question.

■ Problem

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

```
Example:
s = "abcabcbb" → Output: 3 ("abc")
```

■ Core Idea — Sliding Window + Hash Map

Maintain two pointers (`left`, `right`) representing a dynamic window containing unique characters. Use a dictionary (`last`) to store the last seen index of each character. If a repeat is found within the window, move `left` to one position after its previous occurrence.

```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        last = {}          # char -> last index seen
        left = 0           # left end of current window
        max_len = 0

        for right, ch in enumerate(s):
            if ch in last and last[ch] >= left:
                # Duplicate found inside window -> move left pointer
                left = last[ch] + 1
            last[ch] = right
            max_len = max(max_len, right - left + 1)

        return max_len
```

■ Time Complexity: O(n) ■ Space Complexity: O(min(n, charset))

■ Step■by■Step Trace for s = 'abcabcbb'

Step	right	char	left	Window	Action	max_len
1	0	a	0	a	New char	1
2	1	b	0	ab	New char	2
3	2	c	0	abc	New char	3
4	3	a	1	bca	Repeat 'a' → move left	3
5	4	b	2	cab	Repeat 'b' → move left	3
6	5	c	3	abc	Repeat 'c' → move left	3
7	6	b	5	cb	Repeat 'b' → move left	3
8	7	b	7	b	Repeat 'b' → move left	3

■■ Visual Diagram

```
String:  a   b   c   a   b   c   b   b
Index :  0   1   2   3   4   5   6   7
          |-----------|
        left=0     right=2   Window="abc"
        ■ Unique → max_len=3

When 'a' repeats (index 3):
Move left to index 1 (right after old 'a')
          |-----------|
              left=1  right=3  Window="bca"
        ■ Still length 3
```

■ Key Observations

• The window expands when new unique characters are found.

• When duplicates appear, move `left` to one past the last occurrence.

• `max_len` is updated as `right - left + 1` at every step.

■ Complexity

• Time Complexity: O(n) — each character is visited at most twice.

• Space Complexity: O(k) — where k is the number of unique characters.

■ Takeaways

• Sliding window ensures linear time performance.

• Dictionary lookups make duplicate detection O(1).

• Works seamlessly for Unicode strings.

• Key insight: only move `left` forward; never backward.