

# Longest Substring Without Repeating Characters

Classic sliding window — expand greedily, shrink on conflict ■

● MEDIUM

## ■ Hook — Why This Problem Matters

This is one of the most-asked sliding window problems in top tech interviews. Amazon, Google, Microsoft — they all love it. But beyond interviews, it teaches you the core idea behind every sliding window solution you'll ever write. **Master this one, and the pattern clicks for everything else.**

### ■ Real-World Analogy

Imagine you're scanning a barcode reader and you want the longest streak of unique items on a conveyor belt. The moment a duplicate appears, you slide your scanner forward until the duplicate is gone. That's exactly this algorithm.

## ■ Core Intuition — Deep Dive

Brute force would check all  $O(n^3)$  or  $O(n^2)$  substrings. That's too slow. The key insight: **you never need to shrink from scratch**. When you hit a duplicate character at position right, you just move left forward past the previous occurrence of that character. The window is always the 'current valid substring'.

Think of two pointers — **left** and **right** — as the window boundaries. Right always moves forward. Left only moves forward when there's a conflict. A HashMap tracks where each character was last seen.

## ■ Visual Understanding — Step by Step

```
String: "a b c a b c b b"
      0 1 2 3 4 5 6 7

left=0, right=0: 'a' → window=[a]      map={a:0}      len=1
left=0, right=1: 'b' → window=[ab]     map={a:0,b:1}   len=2
left=0, right=2: 'c' → window=[abc]     map={...c:2}   len=3 ★MAX
left=0, right=3: 'a' → 'a' seen at 0! left=max(0,0+1)=1
                    window=[bca]       map={a:3}      len=3
left=1, right=4: 'b' → 'b' seen at 1! left=max(1,1+1)=2
                    window=[cab]       map={b:4}      len=3
left=2, right=5: 'c' → 'c' seen at 2! left=max(2,2+1)=3
                    window=[abc]       map={c:5}      len=3
left=3, right=6: 'b' → 'b' seen at 4! left=max(3,4+1)=5
                    window=[cb]        map={b:6}      len=2
left=5, right=7: 'b' → 'b' seen at 6! left=max(5,6+1)=7
                    window=[b]         len=1

★ ANSWER = 3 (substrings 'abc' at positions 0-2, 1-3, 2-4)
```

### ■ The Subtle Trick — $\max(\text{left}, \text{prevIndex} + 1)$

Why not just do  $\text{left} = \text{map.get}(c) + 1$  directly? Because the previous occurrence might be BEHIND the current left pointer already — meaning it's outside the window. Using  $\max()$  ensures left never moves backward. This is the key insight most beginners miss.

## ■■ Algorithm — Step by Step

- 1 Initialize**  
left=0, maxLen=0. Create HashMap to store last seen index.
- 2 Expand right**  
For each character at index right, check if it exists in the map.
- 3 Conflict? Slide left.**  
If char is in map AND  $\text{map.get}(\text{char}) \geq \text{left}$ , move  $\text{left} = \text{map.get}(\text{char}) + 1$ .
- 4 Update map**  
Always update  $\text{map.put}(\text{char}, \text{right})$  — record the latest position.
- 5 Track max**  
 $\text{maxLen} = \text{Math.max}(\text{maxLen}, \text{right} - \text{left} + 1)$ .

6

### Return maxLen

After processing all characters, return the maximum window size seen.

## ■ Code Implementation (Java)

```
import java.util.HashMap;

public class LongestSubstringNoRepeat {

    public int lengthOfLongestSubstring(String s) {
        HashMap<Character, Integer> map = new HashMap<>();
        int left = 0, maxLen = 0;

        for (int right = 0; right < s.length(); right++) {
            char c = s.charAt(right);

            // If char was seen AND is inside current window
            if (map.containsKey(c) && map.get(c) >= left) {
                left = map.get(c) + 1; // shrink from left
            }

            map.put(c, right); // update latest position
            maxLen = Math.max(maxLen, right - left + 1);
        }

        return maxLen;
    }
}

// Optimized variant using int[128] instead of HashMap (ASCII):
public int lengthOfLongestSubstringFast(String s) {
    int[] index = new int[128]; // last seen index + 1
    int left = 0, maxLen = 0;
    for (int r = 0; r < s.length(); r++) {
        left = Math.max(left, index[s.charAt(r)]);
        maxLen = Math.max(maxLen, r - left + 1);
        index[s.charAt(r)] = r + 1; // store r+1 so max() works directly
    }
    return maxLen;
}
```

## ■■ Time & Space Complexity

### ■ TIME COMPLEXITY

**$O(n)$**

### ■ SPACE COMPLEXITY

**$O(\min(n, m))$**

Time: Each character is visited at most twice (once by right, once by left). Space: HashMap holds at most  $\min(n, \text{charset\_size})$  entries. For ASCII:  $O(128) = O(1)$ . The array optimization reduces space and improves cache performance.

## Common Mistakes

### Not using `max()` when updating left

Doing `left = map.get(c) + 1` without `max()` can move left BACKWARD when a char appears outside the window. This corrupts your window. Always use `left = Math.max(left, map.get(c) + 1)`.

### Forgetting to update map after shrinking

Some beginners skip `map.put(c, right)` when a duplicate is found. You MUST update it — the window now contains the new position of that character.

## Patterns & Connections

This is the foundation of the **Variable-Size Sliding Window** pattern. You'll see it in:

- Longest Substring with At Most K Distinct Characters (same idea, k-limit)
- Fruits Into Baskets (next topic — 2 types =  $k=2$ )
- Minimum Window Substring (shrink when condition met)
- Any problem asking for longest/shortest subarray/substring with a constraint

## Practice Problems

| # | Problem                                         | Difficulty | Pattern                   |
|---|-------------------------------------------------|------------|---------------------------|
| 1 | Longest Substring Without Repeating Characters  | Medium     | Sliding Window + HashMap  |
| 2 | Longest Substring with At Most 2 Distinct Chars | Medium     | Sliding Window + HashMap  |
| 3 | Longest Substring with At Most K Distinct Chars | Medium     | Sliding Window + HashMap  |
| 4 | Find All Anagrams in a String                   | Medium     | Fixed Window + Freq Array |
| 5 | Minimum Window Substring                        | Hard       | Sliding Window + Two Maps |

## Quick Revision

- ✓ Two pointers: left (shrinks window), right (expands window)
- ✓ HashMap stores char → last seen index
- ✓ On duplicate:  $\text{left} = \text{Math.max}(\text{left}, \text{map.get(c)} + 1)$
- ✓ Always update  $\text{map.put(c, right)}$  for every character
- ✓ Answer =  $\text{max}(\text{right} - \text{left} + 1)$  seen during traversal
- ✓ Time:  $O(n)$  | Space:  $O(\text{charset size}) \approx O(1)$  for ASCII

#### ■ MENTAL MODEL — Remember This Forever

Think of left and right as two ends of a rubber band stretched over the string. Right always pulls the band forward. When the band encloses a duplicate, left snaps forward just past the previous copy. The band never shrinks backward. The longest stretch it ever reaches is your answer.