PrepLetter #9

From the Room of a real Interview

Hello, friend!
Welcome to day nine of our coding interview journey. If you’re feeling a little tired, trust me, you’re not alone. Solving three problems every day is no walk in the park. So today, let’s both catch our breath and dive deeply into just *one* classic interview problem.

But don’t think this means we’re slowing down. Today, you’ll learn how to *actually* solve a problem in the style of a real interview: from start to finish, with every step, every thought, and every communication tip covered. Ready? Let’s get you even closer to interview success.

How Coding Interviews Work (and What Interviewers Are Looking For)

Imagine you’re in a virtual room with an interviewer. They give you a problem, and expect you to:

- **Clarify requirements**: Don’t jump to code! Ask questions, restate the problem, make sure you understand what’s being asked.
- **Think out loud**: Share your thoughts, ideas, and worries. Silence is your enemy.
- **Start simple**: Begin with the first solution that comes to mind, even if it’s slow.
- **Optimize**: Once you have a working answer, discuss how to make it better.
- **Code cleanly**: Write code that’s clear, correct, and easy to follow.
- **Test and explain**: Run through test cases out loud. Be your own toughest critic.
- **Handle edge cases**: Show you can think about what could go wrong.
- **Communicate**: Throughout, talk with your interviewer. If you need time to think, say so.

This is what great interviewers look for: not just the right answer, but the right process, attitude, and communication.

So, that's all about today's article, you are tired, I am tired, so we need a break. See you tomorrow!

Ohh wow! you made it here! You really want to learn, don't you? Then let's get into the problem of the day.

The Problem: Longest Substring Without Repeating Characters

Today’s challenge is a staple of interview rooms everywhere:

> **“Given a string s, find the length of the longest substring without repeating characters.”**
> LeetCode link

Let’s see this in action:

```
Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc" with length 3.
```

Interviewer’s Open-Ended Approach

Sometimes, interviewers give you a deliberately vague statement.
Here’s how you *should* react:

Questions to Ask the Interviewer:

- Should the substring be contiguous?

(Usually, yes. But clarify.)

- Does case matter? Is "A" the same as "a"?

(Usually, yes, case matters. But check.)

- What if the string is empty?

(What should we return? Usually 0.)

- Are there non-alphabetic characters?

(Should be handled like any character.)

- Can the input be very long?

(Time complexity might matter.)

The interviewer *wants* you to clarify requirements!

Walk Through an Example

Suppose you’re given:

```
s = "pwwkew"
```

(I know you tried to see how does "pwwkew" sounds!)

Let’s look for the longest substring without repeating characters.

- Start with "p" → "pw" (no repeats) → "pww" (repeat: "w"), so start new substring from next "w": "wk" → "wke" → "wkew" ("w" repeats, so "kew" is valid).
- So, the answer is "wke" or "kew", length 3.

Try this example yourself:
Input: `"bbbbb"`
What do you expect? (Answer: 1, since only "b" is unique.)

Solution 1: Start with a Naive Solution

In interviews, *always* start simple. Don’t jump to the optimal solution. Just give the first solution that comes to mind, even if it’s slow.

Tell your interviewer:

> “The first solution that comes to mind is a naive solution that checks every possible substring... ”

Brute Force Idea

Check every possible substring. For each, check if it has duplicates. It'd be a brute force solution where I'll run two nested loops to generate all substrings and check for uniqueness. But there should be more efficient ways, let me think about it.

Why the above helps?
- The interviewer quickly sees that you can at least solve the problem and you're already thinking about optimizations.
- The interviewer can take a small note "candidate was able to discuss a solution"

See? It's a win!

You might be asked then to discuss more about the brute force solution, about the complexity etc. and, some of the time you might be in luck and the interviewer might say "Great! Let's implement that." And sometimes they can say- "yes that sounds good and how can you optimize it?"

Before we get to the optimal solution, how about you try writing the brute force solution yourself first just for practice?

Pseudocode

```
• For every start index i in s:
• For every end index j > i:
• Check if substring s[i:j] has all unique characters.
• If yes, update max length.
```

Communicate

Let's go for the optimized one now.
You're not a superhero, it's not always expected to come up with the optimal solution right away. So, you might need some time to find out the optimal solution. Maybe you can't take the whole 1 hour to think, but you can take a few minutes to think about it.

Communicate with your interviewer:
> “Let me take a moment to think about how I can optimize this."

> "Is it okay if I use the pen and paper to jot down my thoughts?”"

Interviewer in most cases will say yes, and also they may interrupt you to ask questions or guide you. This is a good sign, it means they are engaged and want to help you succeed. Communicate well with them!

Normally, the interviewer will expect you to come up with a more efficient solution than the brute force one. But we're going to discuss two optimal solutions here anyway. Your task again here is to discuss the optimal solution with the interviewer before you write up the code.
Once the interviewer agrees on the approach, you can start coding after asking them.

Now let's go finding the optimal solution-

Solution 2: Optimize with Sliding Window + Hash Map

Now think:
How can I avoid checking the same substrings over and over?

The **Sliding Window** technique is perfect here.
If you don't know about it, don't worry, it's a common technique used in many problems. It allows us to maintain a window of characters that are currently unique. It uses a left and a right pointer to represent a window and expands or contracts the window based on our requirements.
You can watch a simple video here: Sliding Window Technique.
So, for this, We use two pointers (left and right) to represent the current window, and a data structure (like a set or dictionary) to track characters.

Approach

- Move the right pointer forward one character at a time.
- If the character is not in our set, add it.
- If the character is already in the set, remove characters from the left until there are no duplicates.
- At each step, update max length.

Do you wanna try writing this on your own before looking at the solution?

Python Code (Set Version)

```python
def lengthOfLongestSubstring(s: str) -> int:
    char_set = set()
    left = 0
    max_len = 0
    for right in range(len(s)):
        while s[right] in char_set:
            char_set.remove(s[left])
            left += 1
        char_set.add(s[right])
        max_len = max(max_len, right - left + 1)
    return max_len
```

Complexity

- Space Complexity: O(min(N, M)), where M is the size of the charset (like ASCII or Unicode).
- Time Complexity: O(N), 2N to be exact! since each character is visited at most twice.

Can you think of a way to make it *even better*?

Solution 3: Sliding Window with Hash Map for Indexing

Let’s optimize further.
Instead of just knowing if a character exists, let’s store its *most recent index*.

Approach

- Use a dictionary `char_index` to store the last index of each character.
- When we see a repeating character, we can skip directly to the index after its previous occurrence.

Python Code

```
def lengthOfLongestSubstring(s: str) -> int:
    char_index = {}
    max_len = 0
    left = 0
    for right, char in enumerate(s):
        if char in char_index and char_index[char] >= left:
            # Move left to index after last occurrence of char
            left = char_index[char] + 1
        char_index[char] = right
        max_len = max(max_len, right - left + 1)
    return max_len
```

Time and Space Complexity

- Time: O(N), each character processed once.
- Space: O(min(N, M)), where M is the size of the charset.

Pros and Cons

- Handles large inputs efficiently.
- Slightly more complex, but more optimal.

Once the solution is implemented, try reading through it to see if you've made any mistakes. it's a good practice to read your code out loud, as if explaining it to someone else. This helps catch errors and solidify your understanding.
Many would suggest that- don't tell you interviewer that "I'm done coding right after you're done checking your code", instead tell them- "Looks like it should work, but let me try to dry-run it with a test case to be sure. Do we have time for that?"

Always be open to questions or suggestions from the interviewer. They might have insights or want to see different approaches. Or maybe you just missed a simply silly case and they'll remind you or give you a hint about that, take those points very carefully.

Dry Running the Solution (Interview Style)

Let’s dry-run with `s = "abba"`:

For dry-running, you just take the input, and walk through the code step by step, explaining what happens at each line, how the code is checking a condition, how it's making a decision, etc. This is a great way to catch bugs and ensure your logic is sound.

- Start: left=0, right=0, max_len=0
- right=0, char='a': not seen, set char_index['a']=0, max_len=1
- right=1, char='b': not seen, set char_index['b']=1, max_len=2
- right=2, char='b': seen at 1 >= left=0, move left to 2, set char_index['b']=2, max_len=2
- right=3, char='a': seen at 0 < left=2, so no update to left, set char_index['a']=3, max_len=2

Result: 2 (substring "ab" or "ba")

Test Case for You:
Try running the code on `"dvdf"` by hand.
What should the answer be? (Expected: 3, substring "vdf")

Explaining Your Thought Process

Sometimes you do dry-running before explaining your solution, sometimes after. It depends on the interviewer and the flow of the conversation. But always be ready to explain your thought process.

- What does each part of the code do?
- Why did you choose this approach?
- What is the time and space complexity?
- What edge cases did you consider? (Empty string, all unique chars, all same chars.)

Testing Code:

Try a few edge cases:

```
print(lengthOfLongestSubstring(""))        # Output: 0
print(lengthOfLongestSubstring("a"))       # Output: 1
print(lengthOfLongestSubstring("abcabcbb"))# Output: 3
print(lengthOfLongestSubstring("bbbbb"))   # Output: 1
print(lengthOfLongestSubstring("pwwkew"))  # Output: 3
```

Communicating in the Interview

- **If you’re stuck**: “Let me take a moment to think about this.”
- **When you find a bug**: “I just realized my code doesn’t handle X, let me fix that.”
- **If you finish early**: “Would you like me to discuss further optimizations or possible extensions?”
- **When dry-running**: “Let me walk through my code with a test case to make sure it works as expected.”
- Check multiple times with the interviewer that if you both are on the same page, and if they have any questions or suggestions. This shows you value their input and are open to feedback.

Prepare for Follow-Ups

Interviewers might ask:

- What if you return the substring itself, not just the length?
- How would this work for Unicode characters?
- What if you process the string as a stream?

Summary and Tips

Today, you practiced the *full* interview process:

- Clarify the problem and requirements.
- Start with a simple, brute-force approach.
- Optimize using better data structures and algorithms.
- Code cleanly, comment thoughtfully, and test carefully.
- Communicate at every step.
- Dry-run your solution with examples, but for many companies, they expect you to run the code and fix any bugs on the spot, so be prepared for that as well.

Common Pitfalls:

- Jumping to code before fully understanding the problem.
- Not considering edge cases (like empty strings).
- Not communicating your thought process.
- Forgetting to test your code with different cases.
- Not discussing time and space complexity.

Action List

- Try solving all the approaches above yourself, from brute force to optimal.
- Practice thinking out loud as you solve problems, even when alone.
- Dry-run your code with different test cases; explain each step.
- Explore other problems that use the sliding window technique, like "Minimum Window Substring" or "Longest Subarray with Sum K."
- Practice clarifying requirements by asking questions before you code.
- Review your code for edge cases (empty input, single char, all unique, all duplicates).

You did great today by going deep instead of wide, If I were your interviewer, I would be impressed! Remember, interviews are as much about communication and problem-solving as they are about coding. See you tomorrow for more prep and progress!

Stay curious,

PrepLetter