LeetCode #3 - Longest Substring Without Repeating Characters

October 01, 2020

Hey happy folks 👋! It's a brand new day and it's time to look at another problem from LeetCode - **Longest Substring Without Repeating Characters**.

0003 - Longest Substring Without Repeating Characters.

Problem Statement

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

Constraints:

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

Examples

- Example 1:

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

- Example 2:

```
Input: s = "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.
```

- Example 3:

```
Input: s = "pwwkew"
Output: 3
Explanation: The answer is "wke", with the length of 3.
Notice that the answer must be a substring, "pwke" is a subsequence and not a substri
```

Analysis

The input will be a string containing characters which may or may not be repeated. The ask is to find the longest substring (a part of the given string) having all distinct characters.

One characteristic of a *substring* is that all the characters are contiguous. For e.g, in a given string `s = redquark`, the valid substrings are - `edq`, `ar`, `red`, `quar` etc. The substrings `rd`, `qur` are not valid substrings because even though they contain characters from the source strings, those characters are not continuous.

Apart from above, we are given constraints on the size of string and the characters that are present in a string.

There can be many substrings as outputs but we have to return only the longest among them.

Approach

Prerequisite: Sliding Window Algorithm.

From the input, we can gather the following information -

1. Given data structure is a string which is a linear data structure.
2. The output must be a substring - a part of the given string.
3. Naive solution is to check for each combination of characters in the string

Are you thinking what I am thinking 🤔? Yes, this is a classic example of a problem that can be solved using the legendary technique - ***Sliding Window Algorithm***.

Following are the steps that we will follow -

1. Have two pointers which will define the starting index `start` and ending index `end` of the current window. Both will be `0` at the beginning.
2. Declare a `Set` that will store all the unique characters that we have encountered.
3. Declare a variable `maxLength` that will keep track of the length of the longest valid substring.
4. Scan the string from left to right one character at a time.
5. If the character has not encountered before i.e., not present in the `Set` the we will add it and increment the `end` index. The `maxLength` will be the maximum of `Set.size()` and existing `maxLength` .
6. If the character has encounter before, i.e., present in the `Set` , we will increment the `start` and we will remove the character at `start` index of the string.

7. Steps #5 and #6 are moving the window.
8. After the loop terminates, return `maxLength`.

Time Complexity

We are scanning the string from left to right only once, hence the time complexity will be ***O(n)***.

Space Complexity

We are using `Set` as a data structure to facilitate our computation, therefore, the space complexity should also be O(n), right? Wrong!

WHY?

The problem clearly states that the string contains only English letters, digits, symbols and spaces and are covered in 256 code points. Therefore, a string will be made up of a combination of these characters.

Since a `Set` can contain only unique elements, at any point the size of `Set` cannot be more than 256.

What does this mean? This means that the size of set is a function independent of the size of the input. It is a constant. Therefore, the space complexity will be ***O(1)*** (let me know in comments if you think otherwise).

Code

After analyzing and deciding on the approach, it’s time to write some code -

Java

```java
package org.redquark.tutorials.leetcode;

import java.util.HashSet;
import java.util.Set;

public class LongestSubstringWithoutRepeatingCharacters {

    public int lengthOfLongestSubstring(String s) {
        // Base condition
        if (s == null || s.equals("")) {
            return 0;
        }
        // Starting window index
        int start = 0;
        // Ending window index
        int end = 0;
        // Maximum length of substring
        int maxLength = 0;
        // This set will store the unique characters
        Set<Character> uniqueCharacters = new HashSet<>();
        // Loop for each character in the string
        while (end < s.length()) {
            if (uniqueCharacters.add(s.charAt(end))) {
                end++;
                maxLength = Math.max(maxLength, uniqueCharacters.size());
            } else {
                uniqueCharacters.remove(s.charAt(start));
                start++;
            }
        }
        return maxLength;
    }
}
```

Python

```python
def lengthOfLongestSubstring(s: str) -> int:
    # Base condition
    if s == "":
        return 0
    # Starting index of window
```

```
    start = 0
    # Ending index of window
    end = 0
    # Maximum length of substring without repeating characters
    maxLength = 0
    # Set to store unique characters
    unique_characters = set()
    # Loop for each character in the string
    while end < len(s):
        if s[end] not in unique_characters:
            unique_characters.add(s[end])
            end += 1
            maxLength = max(maxLength, len(unique_characters))
        else:
            unique_characters.remove(s[start])
            start += 1
    return maxLength
```

JavaScript

```
var lengthOfLongestSubstring = function (s) {
    // Base condition
    if (!s) {
        return 0;
    }
    // Starting index of the window
    let start = 0;
    // Ending index of the window
    let end = 0;
    // Maximum length of the substring
    let maxLength = 0;
    // Set to store the unique characters
    const uniqueCharacters = new Set();
    // Loop for each character in the string
    while (end < s.length) {
        if (!uniqueCharacters.has(s[end])) {
            uniqueCharacters.add(s[end]);
            end++;
            maxLength = Math.max(maxLength, uniqueCharacters.size);
        } else {
            uniqueCharacters.delete(s[start]);
            start++;
        }
    }
```

```
    return maxLength;
};
```

Kotlin

```
fun lengthOfLongestSubstring(s: String): Int {
    // Base condition
    if (s == "") {
        return 0
    }
    // Starting window index
    var start = 0
    // Ending window index
    var end = 0
    // Maximum length of substring
    var maxLength = 0
    // This set will store the unique characters
    val uniqueCharacters: MutableSet<Char> = HashSet()
    // Loop for each character in the string
    while (end < s.length) {
        if (uniqueCharacters.add(s[end])) {
            end++
            maxLength = maxLength.coerceAtLeast(uniqueCharacters.size)
        } else {
            uniqueCharacters.remove(s[start])
            start++
        }
    }
    return maxLength
}
```

Conclusion

That’s all folks! In this post, we solved LeetCode problem #3 using **Sliding Window Technique.**

I hope you have enjoyed this post. Feel free to share your thoughts on this.

You can find the complete source code on my GitHub repository. If you like what you learn. feel free to fork 🔪 and star ⭐ it.

- Java
- Python
- JavaScript
- Kotlin

Till next time… Happy coding 😄 and Namaste 🙏!

Created and maintained by **@Anirudh Sharma**

I love to learn and share. Hence, this site has no ads, no affiliation links, or any BS. If you like what you see, give me a thumbs up.

← LeetCode #2 - Add Two Numbers Represented By Linked Lists

LeetCode #4 - Median Of Two Sorted Arrays →

1 Comment - *powered by utteranc.es*

developerlite18 commented a month ago

Hi Guys
I made a simple video explaining the solution using sliding window concept. Please check this out

Icons made by Freepik from www.flaticon.com and Built with Gatsby-starter-bee