1876. Substrings of Size Three with Distinct Characters

Easy Hash Table String Counting Sliding Window

Problem Description

The problem presents a scenario where we have to find substrings of length three without any repeating characters, defined as *good substrings*, within a given string `s`. It's important to remember that substrings are consecutive characters found within `s`.

An important detail is that each occurrence of such a good substring counts separately even if they are the same combinations of characters. So if "abc" occurs five times in different parts of string `s`, it counts as five good substrings.

The goal is to return the total number of such good substrings found in `s`.

Intuition

Coming up with the intuition for solving this problem involves realizing that we can check every substring of length three within the string `s` sequentially.

For each position in the string, starting from the first character and moving towards the last possible position for a substring of length three, we can check whether the three consecutive characters at that position are all different. If they are indeed all unique, then we have identified a good substring.

The approach is to loop through the string starting from the first character and ending two characters before the last one (since we're looking at substrings of length three), and for each position, check the uniqueness of the characters. We increment a count whenever we identify a unique combination, which indicates a good substring.

Since we only need to verify the distinctness of three characters at a time, the checks can be done quickly and efficiently, leading to a solution with linear time complexity, as the number of checks is directly proportional to the length of the string `s`.

Solution Approach

The solution for this problem is implemented using a single for-loop that iterates from index `0` to `n - 2` of the input string `s`, where `n` is the length of the string. This is done to make sure that we can always check substrings of exactly three characters until the end of the string without going out of range.

Inside the loop, for each index `i`, the characters at position `i`, `i + 1`, and `i + 2` are compared to each other. The logic of the comparison is that a substring is good if and only if no two characters out of the three are equal. This is directly translated into the code by the conditional expression:

```
s[i] != s[i + 1] and s[i] != s[i + 2] and s[i + 1] != s[i + 2]
```

If the above expression evaluates to `True`, it means that all three characters are unique, and thus, we found a good substring. The code uses this truthy or falsey value to increment the `count`. Since `True` is equivalent to `1` and `False` is equivalent to `0` in Python, the expression directly contributes to the count.

No additional data structures are used or needed since the task is simply to count instances, and therefore, memory usage is minimal.

The simplicity of this approach is in its linear time complexity (`O(n)`), as it only requires a single pass through the string, checking each set of three characters exactly once. This makes it optimal for strings of any length.

Example Walkthrough

Let's apply the solution approach to a small example to understand how it works. Suppose we have the following string `s`:

```
s = "xyzzabc"
```

The goal is to find good substrings of length three within `s` without any repeating characters.

Now, following the solution approach, we will check each substring of length three:

1. Start with the first substring `"xyz"`. This substring has no repeating characters, making it a good substring. So here, we increment our count to `1`.
2. Move to the next substring `"yzz"`. This contains repeating characters (`'z'`), so it is not a good substring. The count remains `1`.
3. Next, we look at the substring `"zza"`. This also has repeating characters (`'z'`), so it's not a good substring. The count is still `1`.
4. Now, check `"zab"`. All characters are unique here, so this is a good substring. Increase the count to `2`.
5. Finally, we look at `"abc"`, which also contains all unique characters. This is another good substring, so we increment the count to `3`.

After iterating through the string `s`, we found a total of `3` good substrings. Hence, the outcome returned by the solution would be `3`. The check for uniqueness is efficient as we compare only three characters at a time and increment the count based on the condition of uniqueness, leading to a solution with a linear runtime.

Solution Implementation

Python

```python
class Solution:
    def countGoodSubstrings(self, s: str) -> int:
        # Initialize the count of good substrings
        good_substring_count = 0

        # Calculate the length of the string for boundaries in the loop
        string_length = len(s)

        # Loop through the string, stopping at the third-to-last character
        for index in range(string_length - 2):
            # Check if the current character, the next one, and the one after that are all unique
            if s[index] != s[index + 1] and s[index] != s[index + 2] and s[index + 1] != s[index + 2]:
                # If they are unique, we have found a good substring, so increment the count
                good_substring_count += 1

        # After the loop, return the total count of good substrings found
        return good_substring_count
```

Java

```java
class Solution {

    /**
     * This method counts the number of good substrings in a given string.
     * A good substring is defined as a substring with exactly 3 characters and each character is unique.
     *
     * @param s The input string to be searched for good substrings.
     * @return The number of good substrings found.
     */
    public int countGoodSubstrings(String s) {
        int count = 0; // Initialize a counter to store number of good substrings
        int n = s.length(); // Get the length of the input string

        // Loop through the string, up to the third last character
        for (int i = 0; i < n - 2; ++i) {
            // Extract the current, next and the character after next characters
            char firstChar = s.charAt(i);
            char secondChar = s.charAt(i + 1);
            char thirdChar = s.charAt(i + 2);

            // Check if all three characters are distinct
            if (firstChar != secondChar && firstChar != thirdChar && secondChar != thirdChar) {
                // Increment the count if the substring is good
                ++count;
            }
        }

        // Return the total count of good substrings found
        return count;
    }
}
```

C++

```cpp
#include <string>

// Function to count the number of good substrings in a given string.
// A good substring is defined as a substring of length 3 with all unique characters.
int countGoodSubstrings(const std::string& s) {
    // Get the length of the string.
    int length = s.length();
    // Initialize the count of good substrings to zero.
    int goodSubstringCount = 0;

    // Iterate over each character in the string, stopping 2 characters before the end.
    for (int i = 0; i < length - 2; ++i) {
        // Extract the current character and the next two characters.
        char char1 = s[i],
             char2 = s[i + 1],
             char3 = s[i + 2];

        // Check if all three characters are distinct.
        if (char1 != char2 && char1 != char3 && char2 != char3) {
            // If they are, increment the count of good substrings.
            ++goodSubstringCount;
        }
    }

    // Return the total count of good substrings found in the string.
    return goodSubstringCount;
}
```

TypeScript

```typescript
// Function to count the number of good substrings in a given string.
// A good substring is defined as a substring of length 3 with all unique characters.
function countGoodSubstrings(s: string): number {
    // Get the length of the string.
    const length: number = s.length;
    // Initialize the count of good substrings to zero.
    let goodSubstringCount: number = 0;

    // Iterate over each character in the string, stopping 2 characters before the end.
    for (let i: number = 0; i < length - 2; ++i) {
        // Extract the current character and the next two characters.
        let char1: string = s.charAt(i),
            char2: string = s.charAt(i + 1),
            char3: string = s.charAt(i + 2);

        // Check if all three characters are distinct.
        if (char1 !== char2 && char1 !== char3 && char2 !== char3) {
            // If they are, increment the count of good substrings.
            ++goodSubstringCount;
        }
    }

    // Return the total count of good substrings found in the string.
    return goodSubstringCount;
}
```

```python
class Solution:
    def countGoodSubstrings(self, s: str) -> int:
        # Initialize the count of good substrings
        good_substring_count = 0

        # Calculate the length of the string for boundaries in the loop
        string_length = len(s)

        # Loop through the string, stopping at the third-to-last character
        for index in range(string_length - 2):
            # Check if the current character, the next one, and the one after that are all unique
            if s[index] != s[index + 1] and s[index] != s[index + 2] and s[index + 1] != s[index + 2]:
                # If they are unique, we have found a good substring, so increment the count
                good_substring_count += 1

        # After the loop, return the total count of good substrings found
        return good_substring_count
```

Time and Space Complexity

The given Python code snippet defines a method `countGoodSubstrings` which counts the number of substrings of length 3 with all unique characters within a given string `s`.

Time Complexity

To analyze the time complexity, we look at the number of operations that the code performs. The `for` loop runs from `0` to `n - 2`, where `n` is the length of the string `s`. Within each iteration of the loop, there are constant time comparisons being made: checking whether `s[i]`, `s[i + 1]`, and `s[i + 2]` are all unique characters. Since the number of iterations is dependent on the length of the input string, and all operations within the loop are constant time, the time complexity is `O(n)`, where `n` is the length of the input string `s`.

Space Complexity

Considering the space complexity, the code utilizes a fixed amount of extra space: a single integer `count` to keep track of the number of good substrings, and an integer `n` for storing the length of the string. No additional space that grows with the input size is used. Thus, the space complexity is constant, or `O(1)`.