2743. Count Substrings Without Repeating Character

Medium Hash Table String Sliding Window

Leetcode Link

Problem Description

The problem at hand requires us to examine a string `s` that is composed exclusively of lowercase English letters. Our objective is to determine the count of substrings that are considered 'special'. A substring is categorised as 'special' when it does not include any character that appears at least twice. In essence, it must be devoid of any repeating characters. For instance, in the string `"pop"`, the substring `"po"` qualifies as a 'special' substring, while the complete string `"pop"` does not, as the character `'p'` appears more than once.

To be clear, a substring is defined as a consecutive sequence of characters located within a string. For instance, `"abc"` is a substring of `"abcd"`, but `"acd"` is not, since it is not contiguous.

Our task is to deduce the total count of such 'special' substrings within the given string `s`.

Intuition

The solution hinges on the observation that the start of a new 'special' substring is marked by the addition of a non-repeating character, and this substring extends until a character repeats. Once a character repeats, the substring is no longer 'special', so we must adjust our starting index to ensure all following substrings being counted do not contain repeating characters.

Here's the step-by-step approach to arriving at the solution:

1. Use two pointers – say `i` to scan through the string, and `j` to mark the start of the current 'special' substring. Initialize a counter (using `Counter` from Python's `collections` module) to keep track of the occurrences of each character within the current window delimited by `i` and `j`.
2. Iterate through the string. For each character `c` at index `i`, increment its count in the counter.
3. If the count of the current character `c` goes beyond 1, which means it's repeated, we need to advance the `j` pointer to reduce the count back to not more than 1. We do this by moving `j` to the right and decrementing counts of the characters at `j` until `cnt[c]` is at most 1.
4. The number of 'special' substrings that end with the character at index `i` is `i - j + 1`. This is because we can form a 'special' substring by choosing any starting index from `j` up to `i`.
5. Keep adding this count to an accumulator, `ans`, which holds the total count of 'special' substrings.
6. Continue this process until the entire string has been scanned, and return the count `ans`.

By following this approach, we systematically explore all possible 'special' substrings within the string, by expanding and shrinking the window of characters under consideration, always ensuring that no character within the window repeats.

Solution Approach

The implementation for this solution uses a sliding window pattern along with a dictionary (in Python, this is implemented via the `Counter` class from the `collections` module) in order to keep track of the frequency of characters within the current window.

Here's a step-by-step explanation of the code:

- `cnt = Counter()`: We initialize a `Counter` to maintain the frequency of each character in the current window.
- `ans = 0`: This is our accumulator for the total count of 'special' substrings.
- `j = 0`: This is the starting index of our sliding window.

We go through the string using a `for` loop.

```
for i, c in enumerate(s):
    cnt[c] += 1
```

With each iteration, we:

1. Increment the counter for the current character `c`.
2. Then, we enter a `while` loop which runs as long as the current character's count is more than 1, indicating a repeat.

```
while cnt[c] > 1:
    cnt[s[j]] -= 1
```

Inside the loop:

- We decrement the count of the character at the current start of our window `j`, effectively removing it from our current consideration.
- We move our window start `j` forward by one.

This loop helps maintain the invariant that our sliding window `[j, i]` only contains non-repeating characters.

3. After ensuring no duplicates in the window `[j, i]`, we update our answer with the number of new 'special' substrings ending at `i`:

```
ans += i - j + 1
```

- Here, `i - j + 1` represents the number of 'special' substrings that can be formed where the last letter is at index `i`. This is because any substring starting from `j` to `i` up to this point is guaranteed to be 'special'.

4. Finally, after the loop finishes, we return `ans`, which now contains the total count of all 'special' substrings.

This algorithm uses the sliding window pattern, which is efficient and elegantly handles the continuous checking of the substrings by maintaining a valid set of characters between the `i` and `j` pointers. The use of a `Counter` abstracts away the low-level details of frequency management and provides easy and fast updates for the frequencies of characters in the current window.

Example Walkthrough

Let's consider a small example to illustrate the solution approach.

Assume the string `s` is `"ababc"`.

Here's how the algorithm works for this string:

1. Initialize `cnt` as an empty `Counter`.
2. Initialize `ans = 0` and `j = 0`.
3. Iterate over each character of the string, with `i` being the position in the loop.
   - For `i = 0` (`c = 'a'`):
     - `cnt['a']` becomes 1.
     - No characters are repeated, so `ans` becomes `0 + 1 = 1`.
   - For `i = 1` (`c = 'b'`):
     - `cnt['b']` becomes 1.
     - Still no repeated characters, so `ans` becomes `1 + 2 = 3` (substrings `"ab"` and `"b"`).
   - For `i = 2` (`c = 'a'`):
     - `cnt['a']` becomes 2 (as 'a' is encountered again).
     - `cnt['a']` is greater than 1, so we increment `j` to 1 and decrement `cnt['a']` by 1, making it 1.
     - Update `ans` to become `3 + 2 = 5` (new substrings `"aba"`, `"ba"`).
   - For `i = 3` (`c = 'b'`):
     - `cnt['b']` becomes 2.
     - `cnt['b']` is more than 1, so we increment `j` to 2 and decrement `cnt['b']` to 1.
     - Update `ans` to become `5 + 2 = 7` (new substrings `"ab"`, `"b"`).
   - For `i = 4` (`c = 'c'`):
     - `cnt['c']` becomes 1.
     - No repeated characters, so `ans` becomes `7 + 3 = 10` (substrings `"abc"`, `"bc"`, `"c"`).
4. Once we finish iterating, we end up with `ans = 10`, which is the total count of 'special' substrings.

By following these steps, we can see how the sliding window keeps a valid range by updating the `j` index whenever a repeat character is found and how the count of `ans` is calculated based on the positions of `i` and `j`.

Python Solution

```
from collections import Counter

class Solution:
    def numberOfSpecialSubstrings(self, s: str) -> int:
        # Initialize a counter to keep track of the frequency of letters
        char_counter = Counter()

        # 'total_special_substrings' will hold the count of all special substrings
        total_special_substrings = 0

        # 'start_index' is the index at which the current evaluation of the substring starts
        start_index = 0

        # Iterate over the string, with 'i' as the current index and 'char' as the current character
        for i, char in enumerate(s):
            # Update the frequency of the current character in the counter
            char_counter[char] += 1

            # If the frequency of the current character is more than one,
            # increment the start_index and decrement the frequency of the
            # character at start_index to ensure we are checking for a special substring
            while char_counter[char] > 1:
                char_counter[s[start_index]] -= 1
                start_index += 1

            # A special substring is one where all characters are unique. Since we
            # move the 'start_index' to maintain unique characters in the substring,
            # the difference 'i - start_index + 1' gives us the number of new unique
            # special substrings ending at index 'i'.
            total_special_substrings += i - start_index + 1

        # Return the total count of special substrings found
        return total_special_substrings
```

Java Solution

```
class Solution {

    // Method to count the number of special substrings
    // A special substring consists of a unique character
    public int numberOfSpecialSubstrings(String s) {
        int n = s.length(); // Length of the string
        int specialSubstrCount = 0; // Counter for special substrings
        int[] charCount = new int[26]; // Count array for each character 'a'-'z'

        // Using two pointers, 'start' and 'end', to define the current substring
        for (int start = 0, end = 0; end < n; ++end) {
            int currentCharIdx = s.charAt(end) - 'a'; // Convert current character to index
            // Increase the count for the current character
            ++charCount[currentCharIdx];

            // If there is more than one occurrence of the character, move 'end' forward
            // to reduce the count of the character at the 'end' pointer
            while (charCount[currentCharIdx] > 1) {
                --charCount[s.charAt(end++) - 'a'];
            }

            // The number of special substrings that end at 'start' equals 'start' - 'end' + 1
            // because all substrings between 'end' and 'start' (inclusive) are special
            specialSubstrCount += start - end + 1;
        }

        return specialSubstrCount; // Return the total count of special substrings
    }
}
```

C++ Solution

```
class Solution {
public:
    // This method counts special substrings within a given string.
    // A special substring is defined as a substring with characters occurring only once.
    int numberOfSpecialSubstrings(string s) {
        int n = s.size(); // Length of the string
        int count[26] = {}; // Array to count occurrences of each character
        int answer = 0; // Total count of special substrings

        // Two pointers, 'i' for the current end of substring
        // and 'j' for the beginning of the current special substring
        for (int i = 0, j = 0; i < n; ++i) {
            int charIndex = s[i] - 'a'; // Convert current character to index (0-25)
            ++count[charIndex]; // Increment the count for this character

            // Ensure the current character only appears once
            while (count[charIndex] > 1) {
                // If it appears more than once, move the start pointer 'j' forward
                --count[s[j++] - 'a'];
            }

            // Add the length of the current special substring to the total
            // The length is 'i - j + 1' for substring from 'j' to 'i' inclusive
            answer += i - j + 1;
        }

        return answer; // Return the total number of special substrings
    }
};
```

Typescript Solution

```
function numberOfSpecialSubstrings(s: string): number {
    // Helper function to get the index of a character 'a' to 'z' as 0 to 25.
    const getCharIndex = (char: string) => char.charCodeAt(0) - 'a'.charCodeAt(0);

    // Length of the input string
    const lengthOfString = s.length;

    // Array to store the count of each character
    const charCount: number[] = Array(26).fill(0);

    // Initialize the answer to count the number of special substrings
    let specialSubstringsCount = 0;

    // Two pointers for the sliding window approach
    for (let i = 0, j = 0; i < lengthOfString; ++i) {
        // Get the index of the current character
        const currentIndex = getCharIndex(s[i]);

        // Increment the count for this character
        ++charCount[currentIndex];

        // Ensure that we have at most one of each character in the current sliding window
        while (charCount[currentIndex] > 1) {
            // If more than one, decrement the count of the leftmost character
            --charCount[getCharIndex(s[j++])];
        }

        // The number of special substrings ending at the current position 'i' is the width of the current window
        specialSubstringsCount += i - j + 1;
    }

    // Return the total count of special substrings
    return specialSubstringsCount;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the algorithm is determined by the for-loop and the while-loop inside of it.

- The for-loop runs `n` times, where `n` is the length of the string `s`. In each iteration, the algorithm performs a constant amount of work by updating the `Counter` object and calculating the `ans`.
- The while-loop can execute more than once per for-loop iteration. However, each character from the string `s` can only cause at most two operations on the `Counter` one increment and one decrement. Therefore, despite being nested, the while-loop won't result in more than `2n` operations overall due to the two-pointer approach.

Combining these observations, the for-loop complexity `O(n)` and the while-loop total complexity `O(n)`, we have a total time complexity of `O(n + n) = O(n)`.

Space Complexity

For space complexity, the principal storage consumer is the `Counter` object, which at most will contain a number of keys equal to the number of distinct characters in the string `s`.

- If the alphabet size is constant and small relative to `n` (such as the English alphabet of 26 letters), the space complexity can be considered `O(1)`.
- In a broader perspective where the alphabet size is not constant or the number of distinct characters is proportional to `n`, the space complexity is `O(k)`, where `k` is the number of distinct characters in the string `s`.

Considering the most general case, the space complexity is `O(k)`.