1100. Find K-Length Substrings With No Repeated Characters

Medium Hash Table String Sliding Window

Leetcode Link

Problem Description

Given a string `s` and an integer `k`, the task is to count how many substrings of length `k` exist within `s` that contain no repeated characters. A substring is a contiguous sequence of characters within a string. For example, in the string `abcde`, `abc`, `bcd`, and `cde` are substrings of length `3`. We are specifically interested in substrings where every character is unique, meaning no character appears more than once in that substring.

Intuition

The intuition behind the solution is to use the "sliding window" technique. This approach involves maintaining a window of characters that can expand or shrink as needed while scanning the string from left to right. Here's how the intuition develops:

1. If `k` is greater than the length of the string or if `k` is greater than 26 (the number of letters in the English alphabet), there can't be any valid substrings of length `k` with all unique characters, so we return 0 immediately.
2. Starting at the beginning of the string, we expand our window to include characters until we hit a size of `k` or we find a duplicate character.
3. To efficiently keep track of the characters inside our current window and their counts, we use a `Counter` (a type of dictionary or hashmap in Python), incrementing the count for each new character we add to the window.
4. If at any point a character's count exceeds 1 (meaning we've found a duplicate), or the window's size exceeds `k`, we shrink the window from the left by removing characters and decrementing their respective counts.
5. Every time our window size becomes exactly `k` and all characters are unique (i.e., no count is greater than 1), we have found a valid substring. We then increment the counter for the final answer (`ans`).

Notice that for each position `i` in the string, we adjust the left side of our window (`j`) to ensure the window size does not exceed `k` and there are no repeating characters. The solution iteratively keeps account of all such substrings and returns the count as the final result.

Solution Approach

The solution employs the sliding window approach along with a `Counter` from the collections module to keep track of the number of occurrences of each character within the current window. Here's a step-by-step breakdown:

1. **Initialize Variables**: A counter named `cnt` is used to count occurrences of characters, `ans` for storing the total count of valid substrings found, and `j` for marking the start position of the sliding window.
2. **Early Termination Checks**: Check if `k` is larger than the string length `n` or greater than 26. If so, we return `0` because it's not possible to have substrings of length `k` with unique characters.
3. **Iterate Over the String**: Use a for-loop with `enumerate(s)` to get both index `i` and character `c`.
4. **Expand the Window**: Increase the count of the current character `c` in `cnt`.
5. **Shrink the Window if Necessary**: a. While the count of character `c` in `cnt` is more than `1` (meaning `c` has appeared before in the current window), decrease the count of the leftmost character `s[j]` and increase `j` to move the window's left edge to the right. b. Also, shrink the window if the current window length (`i - j + 1`) exceeds `k`.
6. **Check Window Validity**: If the window length is exactly `k`, and all characters in the window are unique (the while loop has ensured that the counts are `1`), increment `ans` which signifies a valid substring.
7. **Return Result**: After finishing the loop, the `ans` variable contains the number of substrings with length `k` and no repeating characters, which is returned as the solution.

This solution uses a dynamic slide of the window that only moves the left bound forward (`j`), and never backward, ensuring that the algorithm's time complexity is O(n), as each character is processed at most twice (once when added to the window, once when removed). This makes the algorithm efficient for this problem.

Example Walkthrough

Let's illustrate the solution approach with a concrete example. Consider the string `s = "abcba"` and `k = 3`. We want to find the total number of valid substrings of length 3 with no repeating characters.

1. **Initialize Variables**: Initialize `cnt = Counter()`, `ans = 0`, `j = 0`.
2. **Early Termination Checks**: Length of `s` is 5, and `k` is 3, which is less than 26, so we move on without terminating early.
3. **Iterate Over the String**: We start a for-loop that iterates over each character.
   - On the first iteration `i = 0`, `c = 'a'`. We add 'a' to the counter.
   - Then, the window size is `1`, which is less than `3`. Since there are no duplicates, we continue.
4. On the second iteration `i = 1`, `c = 'b'`. We add 'b' to the counter.
   - Now the window size is `2`, which is still less than `3`. No duplicates so, we continue.
5. In the third iteration `i = 2`, `c = 'c'`. We add 'c' to the counter.
   - Our window is now `abc` (`i - j + 1 == 3`), which is equal to `k` and contains no repeating characters.
   - We found a valid substring, so `ans = ans + 1`.
6. Fourth iteration `i = 3`, `c = 'b'`. Counter for 'b' becomes 2, indicating a duplicate.
   - We proceed to shrink the window. We reduce the count of 'a' since it's the leftmost character in the current window and move `j` from `0` to `1`.
   - The window length is again `3` after adjusting `j`, but now it's `bcb` which is not valid because it contains repeating characters 'b'.
7. Fifth iteration `i = 4`, `c = 'a'`. The counter for 'a' is now 2, another duplicate.
   - Continuing to shrink the window, we decrease the count for 'b' and increment `j` to `2`.
   - The window size (`i - j + 1 = 3 - 2 + 1`) now becomes 2. Since it's less than `k`, we continue to expand the window in the next iterations.
8. At this point, the substring `cba` from the previous steps is no longer in our current window. We have a window of size `2` with unique characters `ck`. Unfortunately, this is the last iteration, and there are no more characters in `s` to consider.
9. **Return Result**: No other valid substrings of length `3` are found, so the final `ans` remains `1`.

In conclusion, there is only one valid substring of length `3` with all unique characters in the string `abcba`, which is `abc`. Hence the function would return `1`.

Python Solution

```python
from collections import Counter

class Solution:
    def numKLenSubstrNoRepeats(self, string: str, k: int) -> int:
        # Calculate the length of the input string
        length_of_string = len(string)

        # If the required substring length k is greater than the string length
        # or greater than the alphabet count, there can be no valid substrings.
        if k > length_of_string or k > 26:
            return 0

        # Initialize the answer to 0 and the start index of the current substring to 0
        num_of_substrings = current_start_index = 0

        # Create a Counter to store the frequency of characters in the current substring
        char_freq = Counter()

        # Iterate through the string using the index and character
        for current_end_index, char in enumerate(string):
            # Increment the frequency of the current character
            char_freq[char] += 1

            # While there are duplicate characters in the current substring,
            # or the substring length exceeds k, shrink the substring from the left.
            while char_freq[char] > 1 or current_end_index - current_start_index + 1 > k:
                char_freq[string[current_start_index]] -= 1
                current_start_index += 1

            # If the current substring length is exactly k, we found a valid substring.
            if current_end_index - current_start_index + 1 == k:
                num_of_substrings += 1

        # Return the total count of valid substrings
        return num_of_substrings
```

Java Solution

```java
class Solution {

    // Method to count the number of k-length substrings with no repeating characters
    public int numKLenSubstrNoRepeats(String s, int k) {

        // Get the length of the string
        int stringLength = s.length();

        // If k is greater than the string length or the maximum possible unique characters, return 0
        if (k > stringLength || k > 26) {
            return 0;
        }

        // Create an array to store the count of characters
        int[] charCount = new int[128];

        // Initialize a variable to store the count of valid substrings
        int validSubstrCount = 0;

        // Initialize two pointers for the sliding window technique
        for (int start = 0, end = 0; end < stringLength; ++end) {

            // Increment the count of the current character
            charCount[s.charAt(end)]++;

            // If a character count is greater than 1, or the window size is greater than k,
            // then slide the window from the start
            while (charCount[s.charAt(end)] > 1 || end - start + 1 > k) {
                // Decrement the count of the starting character as it is no longer in the window
                charCount[s.charAt(start++)]--;
            }

            // If the size of the window equals k, we have a valid substring
            if (end - start + 1 == k) {
                validSubstrCount++;
            }
        }

        // Return the total count of valid substrings
        return validSubstrCount;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int numKLenSubstrNoRepeats(string s, int k) {
        int strLength = s.size();

        // If the length of the substring `k` is greater than the string length
        // or there are more characters than the alphabet size, return 0
        // because no such substrings can exist.
        if (k > strLength || k > 26) {
            return 0;
        }

        // Array to keep count of character occurrences
        int charCount[128] = {1}; // Initialize all elements to 0
        int validSubstrCount = 0; // Counter for the number of valid substrings

        // Use a sliding window defined by the range [windowStart, windowEnd)
        for (int windowEnd = 0, windowStart = 0; windowEnd < strLength; ++windowEnd) {
            // Increase the count for the current character at windowEnd
            ++charCount[s[windowEnd]];

            // If there is a duplicate character or the window size exceeds k,
            // shrink the window from the left by increasing windowStart.
            while (charCount[s[windowEnd]] > 1 || windowEnd - windowStart + 1 > k) {
                --charCount[s[windowStart++]];
            }

            // If the window size is exactly k, we found a valid substring without repeating characters.
            // Increase the count of valid substrings.
            validSubstrCount += (windowEnd - windowStart + 1) == k;
        }

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

Typescript Solution

```typescript
function numKLenSubstrNoRepeats(s: string, k: number): number {
    // Calculate the length of the input string.
    const stringLength = s.length;

    // If the requested substring length is greater than the string length, return 0.
    if (k > stringLength) {
        return 0;
    }

    // Map to store the frequency of characters in the current window.
    const frequencyMap: Map<string, number> = new Map();

    // Initialize the frequency map with the first 'k' characters of the string.
    for (let index = 0; index < k; ++index) {
        frequencyMap.set(s[index], (frequencyMap.get(s[index]) ?? 0) + 1);
    }

    // Count valid substrings. A valid substring has no repeated characters.
    let validSubstringCount = frequencyMap.size === k ? 1 : 0;

    // Iterate over the string, starting from the 'k'th character.
    for (let index = k; index < stringLength; ++index) {
        // Add the current character to the map.
        frequencyMap.set(s[index], (frequencyMap.get(s[index]) ?? 0) + 1);
        // Remove or decrement the character count 'k' positions behind.
        frequencyMap.set(s[index - k], (frequencyMap.get(s[index - k]) ?? 0) - 1);

        // If the count of the old character drops to zero, remove it from the map.
        if (frequencyMap.get(s[index - k]) === 0) {
            frequencyMap.delete(s[index - k]);
        }

        // If the current window has exactly 'k' unique characters, increment the result.
        validSubstringCount += frequencyMap.size === k ? 1 : 0;
    }

    // Return the total count of valid substrings.
    return validSubstringCount;
}
```

Time and Space Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the string `s`. This is because the code uses a sliding window approach, traversing the string once. Each character is added to the sliding window, and if a condition is met (more than one occurrence of the same character, or the window size exceeds `k`), the window adjusts by removing characters from the start. The number of adjustments made is proportional to `n` as well.

The space complexity of the code is `O(k)` if `k < 26`, or `O(26)` (which simplifies to `O(1)`) if `k >= 26`, due to the length of the `Counter` dictionary never having more characters than the alphabet case (`k` characters when `k < 26`, and 26 characters of the alphabet when `k >= 26` because no more than 26 unique characters can be in the string at a time given it consists of just the alphabet).