159. Longest Substring with At Most Two Distinct Characters

Medium Hash Table String Sliding Window

Leetcode Link

Problem Description

The problem provides us with a string `s` and asks us to determine the length of the longest substring which contains at most two distinct characters. For example, in the string "aabacbebebe," the longest substring with at most two distinct characters is "cbebebe," which has a length of 7.

The challenge here is to make sure we can efficiently figure out when we have a valid substring (with 2 or fewer distinct characters) and how to effectively measure its length. We are required to iterate through our string and somehow keep track of the characters we have seen, all the while being able to update and retrieve the length of the longest qualifying substring.

Intuition

The intuition behind the solution stems from the two-pointer technique, also known as the sliding window approach. The main idea is to maintain a dynamic window of characters in the string which always satisfies the condition of having at most two distinct characters.

In this solution, we use dictionary `cnt` that acts as our counter for each character within our current window and variables `j` and `ans` which represent the start of the window and the answer (length of longest substring), respectively.

Here's how we can logically step through the solution:

1. We iterate through the string with a pointer `i`, which represents the end of our current window.
2. As we iterate, we update the counter `cnt` for the current character.
3. If ever our window contains more than two distinct characters, we need to shrink it from the start (pointer `j`). We do this inside the `while` loop until we again have two or fewer distinct characters.
4. Inside the `while` loop, we also decrement the count of the character at the start of the window and if its count hits zero, we remove it from our counter dictionary.
5. After we ensure our window is valid, we update our answer `ans` with the maximum length found so far, which is the difference between our pointers `i` and `j` plus one.
6. Finally, we continue this process until the end of the string and return the answer.

This solution is efficient as it only requires a single pass through the string, and the operations within the sliding window are constant time on average, offering a time complexity of O(n), where n is the length of the string.

Solution Approach

The implementation includes the use of a sliding window strategy and a hash table (Counter) for tracking character frequencies. Let's go step by step through the solution:

1. **Initialize the Counter and Variables:**
   - A `Counter` object `cnt` from the `collections` module is used to monitor the number of occurrences of each character within the current window.
   - Two integer variables `ans` and `j` are initialized to 0. `ans` will hold the final result, while `j` will indicate the start of the sliding window.
2. **Iterate Through the String:**
   - The string `s` is looped through with the variable `i` acting as the end of the current window and `c` as the current character in the loop.
   - For each character `c`, its count is incremented in the `Counter` by `cnt[c] += 1`.
3. **Validate the Window:**
   - A `while` loop is utilized to reduce the window size from the left if the number of distinct characters in the `Counter` is more than two.
   - Inside the loop:
     - Decrement the count of the character at the start `s[j]` of the window.
     - If the count of that character becomes zero, it is removed from the Counter to reflect that it's no longer within the window.
     - Increment `j` to effectively shrink the window from the left side.
4. **Update the Maximum Length Result:**
   - After the window is guaranteed to contain at most two distinct characters, the maximum length `ans` is updated with the length of the current valid window, calculated as `i - j + 1`.
5. **Return the Result:**
   - Once the end of the string is reached, the loop terminates, and `ans`, which now holds the length of the longest substring containing at most two distinct characters, is returned.

This algorithm effectively maintains a dynamic window of the string, ensuring its validity with respect to the distinct character constraint and updating the longest length found. The data structure used (`Counter`) immensely simplifies frequency management and contributes to the efficiency of the solution. The pattern used (sliding window) is particularly well-suited for problems involving contiguous sequences or substrings with certain constraints, as it allows for an O(n) time complexity traversal that accounts for all possible valid substrings.

Example Walkthrough

Let's illustrate the solution approach with a smaller example using the string "aabbcca."

1. **Initialize the Counter and Variables:**
   - Initialize `cnt` as an empty `Counter` object and variables `ans` and `j` to 0.
2. **Iterate Through the String:**
   - Set `i` to 0 (starting at the first character 'a').
   - Increment `cnt['a']` by 1. Now, `cnt` has {'a': 1}.
3. **Validate the Window:**
   - No more than two distinct characters are in the window, so we don't shrink it yet.
4. **Update the Maximum Length Result:**
   - Set `ans` to `i - j + 1`, which is `0 - 0 + 1 = 1`.

Next, move `i` to the next character (another 'a') and repeat the steps:

- `cnt['a']` becomes 2, the window still has only one distinct character.
- Update `ans` to `i - j + 1`, which becomes `1 - 0 + 1 = 2`.

Continuing this process:

- For `i = 2` and `i = 3`, the steps are similar as we continue to read 'b'. `cnt` becomes {'a': 2, 'b': 2} and `ans` now is 4.
- At `i = 4`, the first 'c' is encountered, `cnt` becomes {'a': 2, 'b': 2, 'c': 1}, and now `ans` is also updated to 4.
5. **Validate the Window:**
   - With `i = 5`, we encounter the second 'c' and now `cnt` is {'a': 2, 'b': 2, 'c': 2}, but since there are already two distinct characters ('a' and 'b'), the `while` loop will trigger to shrink the window.
   - In the `while` loop, we decrement `cnt['a']` (as 'a' was at the start) and increment `j`; after decrementing twice, 'a' is removed from `cnt`.
   - Now `cnt` is {'b': 2, 'c': 2}, and `j` moves past the initial 'a's, and is 2.
6. **Update the Maximum Length Result:**
   - Update `ans` to `i - j + 1`, which is `5 - 2 + 1 = 4`.

For the last character 'a' at `i = 6`:

- Increment `cnt['a']` to 1, `cnt` becomes {'b': 2, 'c': 2, 'a': 1}.
- Then the `while` loop is entered and 'b's are removed similar to the 'a's before.
- `j` is moved to 4, right after the last 'b'.
- Update `ans` to `i - j + 1`, which is now `6 - 4 + 1 = 3`.
5. **Return the Result:**
   - The string is fully iterated, and the maximum `ans` was 4. Thus, the longest substring with at most two distinct characters is "aabb" or "bbcc", both with a length of 4.

Python Solution

```python
from collections import Counter

class Solution:
    def lengthOfLongestSubstringTwoDistinct(self, s: str) -> int:
        # Initialize a counter to keep track of character frequencies
        char_freq = Counter()

        # Initialize the max_length to keep record of the longest substring with at most 2 distinct characters
        max_length = 0

        # Initialize the start index of the current substring with at most 2 distinct characters
        start_index = 0

        # Iterate over the string using an end_index pointer
        for end_index, char in enumerate(s):
            # Increment the frequency of the current character
            char_freq[char] += 1

            # If the number of distinct characters is more than 2,
            # shrink the window from the left until we have at most 2 distinct characters
            while len(char_freq) > 2:
                char_freq[s[start_index]] -= 1
                # If the count of the leftmost character is now zero, remove it from the counter
                if char_freq[s[start_index]] == 0:
                    del char_freq[s[start_index]]
                # Move left boundary of the window to the right
                start_index += 1

            # Update the max_length with the maximum between the current max_length and
            # the length of the current substring with at most 2 distinct characters
            max_length = max(max_length, end_index - start_index + 1)

        # Return the maximum length of substring found
        return max_length
```

Java Solution

```java
class Solution {
    public int lengthOfLongestSubstringTwoDistinct(String s) {
        // Create a HashMap to store the frequency of each character.
        HashMap<Character, Integer> charFrequencyMap = new HashMap<>();
        int length = s.length();
        int maxLength = 0; // This will hold the length of the longest substring with at most two distinct characters.

        // Two pointers defining the window of characters under consideration
        for (int left = 0, right = 0; right < length; right++) {
            // Get the current character from the string.
            char currentChar = s.charAt(right);
            // Increase the frequency count for the character in our map.
            charFrequencyMap.put(currentChar, charFrequencyMap.getOrDefault(currentChar, 0) + 1);

            // If the map contains more than two distinct characters, shrink the window from the left
            while (charFrequencyMap.size() > 2) {
                char leftChar = s.charAt(left);
                // Decrease the frequency count of this character.
                charFrequencyMap.put(leftChar, charFrequencyMap.get(leftChar) - 1);
                // Remove the character from the map if its count drops to zero, to maintain at most two distinct characters.
                if (charFrequencyMap.get(leftChar) == 0) {
                    charFrequencyMap.remove(leftChar);
                }
                // Move the left pointer to the right
                left++;
            }

            // Calculate the maximum length encountered so far.
            maxLength = Math.max(maxLength, right - left + 1);
        }

        // Return the maximum length found.
        return maxLength;
    }
}
```

C++ Solution

```cpp
#include <unordered_map>
#include <string>
#include <algorithm>

class Solution {
public:
    int lengthOfLongestSubstringTwoDistinct(std::string s) {
        std::unordered_map<char, int> charFrequency; // Map to store the frequency of each character
        int stringSize = s.size(); // Size of the input string
        int maxLength = 0;         // Variable to store the max length so far

        // Two pointers technique, where 'left' is the start of the window and 'i' is the end
        for (int left = 0, i = 0; i < stringSize; ++i) {
            // Increment the frequency of the current character
            // If our map has more than two distinct characters, shrink the window from the left
            while (charFrequency.size() > 2) {
                charFrequency[s[left]] -= 2; // Decrease the frequency of the leftmost character
                if (charFrequency[s[left]] <= 0) { // If frequency is zero, remove it from the map
                    charFrequency.erase(s[left]);
                }
                ++left; // Move the left boundary of the window to the right
            }

            // Calculate the current length of the substring and update the max length
            maxLength = std::max(maxLength, i - left + 1);
        }
        return maxLength; // Return the length of the longest substring with at most two distinct characters
    }
};
```

Typescript Solution

```typescript
// Importing the Map object from the 'es6' library
import { Map } from "es6";

// Define the method lengthOfLongestSubstringTwoDistinct
function lengthOfLongestSubstringTwoDistinct(s: string): number {
    // Create a Map to store the frequency of each character
    let charFrequency = new Map<string, number>();
    // Determine the size of the input string
    const stringSize = s.length;
    // Variable to store the maximum length of substring found so far
    let maxLength = 0;

    // Use the two pointers technique, with 'left' as the start of the window and 'i' as the end
    for (let left = 0, i = 0; i < stringSize; ++i) {
        // Get the current character at index 'i'
        const currentChar = s[i];
        // Increment the frequency of the current character in the Map
        charFrequency.set(currentChar, (charFrequency.get(currentChar) || 0) + 1);

        // If there are more than two distinct characters, shrink the window from the left
        while (charFrequency.size > 2) {
            // Decrease the frequency of the character at the 'left' pointer
            const leftFrequency = charFrequency.get(s[left]) || 0;
            charFrequency.set(s[left], leftFrequency - 1);

            if (leftFrequency > 0) {
                // If frequency is not zero, update it in the Map
                charFrequency.set(s[left], leftFrequency);
            } else {
                // If frequency is zero, remove it from the Map
                charFrequency.delete(s[left]);
            }
            // Move the left boundary of the window to the right
            ++left;
        }

        // Calculate the current length of the substring
        const currentLength = i - left + 1;
        // Update the maximum length if the current substring is found
        maxLength = Math.max(maxLength, currentLength);
    }

    // Return the length of the longest substring with at most two distinct characters
    return maxLength;
}

// Example usage:
// const result = lengthOfLongestSubstringTwoDistinct("eceba");
// console.log(result); // Output would be 3, for the substring "ece"
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is `O(n)`, where `n` is the length of the string `s`. Here's a breakdown of the complexity:

- The `for` loop runs for every character in the string, which contributes to `O(n)`.
- Inside the loop, the while loop might seem to add complexity, but it will not exceed `O(n)` over the entire runtime of the algorithm, because each character is added once to the Counter and potentially removed once when the distinct character count exceeds 2. Thus, each character results in at most two operations.
- The operations inside the while loop, like decrementing the Counter and conditional removal of an element from the Counter, are `O(1)` operations since Counter in Python is implemented as a dictionary.

Hence, combining these, we get a total time complexity of `O(n)`.

Space Complexity

The space complexity of the code is `O(k)`, where `k` is the size of the distinct character set that the Counter can hold at any time. More precisely:

- Since the task is to find the longest substring with at most two distinct characters, the Counter `cnt` will hold at most 2 elements plus 1 element that will be removed once we exceed the 2 distinct characters. So in this case, k = 3. However, k is a constant here, so we often express this as `O(1)`.
- The variables `ans`, `j`, and `c` are constant-size variables and do not scale with the input size, so they contribute `O(1)`.

Therefore, the overall space complexity is `O(1)`, since the Counter size is bounded by the small constant value which does not scale with `n`.