1759. Count Number of Homogenous Substrings

Medium Math String Leetcode Link

Problem Description

The task is to find the number of homogenous substrings in a given string `s`. A string is considered homogenous if all characters in the string are identical. A substring refers to a consecutive sequence of characters within a string. The answer should be returned modulo `10^9 + 7` to prevent overflow issues due to potentially large numbers.

For example, if `s = "aaa"`, then the homogenous substrings are `"a"`, `"a"`, `"a"`, `"aa"`, `"aa"`, and `"aaa"`, which totals up to 6.

Intuition

To solve this problem efficiently, we can utilize a two-pointer technique.

1. We iterate over the string using two pointers, `i` and `j`.
2. The first pointer `i` marks the start of a potential homogenous substring, while `j` scans ahead to find where this substring ends (i.e., where a character different from `s[i]` is encountered).
3. For each character position `i`, we find the longest stretch of the same character by incrementally increasing `j` as long as `s[j]` is equal to `s[i]`.
4. The length of the homogenous substring starting at `i` is `(j - i)`. For each such substring, we calculate the number of homogenous substrings that can be made, which is given by the formula `(1 + cnt) * cnt / 2`, where `cnt` is the length of the homogenous substring.
5. Why this formula? Consider a homogenous string of length `n`. We can make `n` single-character substrings, `n-1` substrings of length 2, `n-2` of length 3, and so on, down to 1 substring of length `n`. This forms an arithmetic sequence that sums to `n*(n+1)/2`.
6. The answer is incremented by this count for each homogenous stretch we find.
7. We use the modulo operation to keep our calculations within the prescribed limit to avoid integer overflow.
8. The first pointer `i` is then set to `j` to start searching for the next homogenous substring.

This approach optimizes the process by minimizing the number of times we traverse the string, leading to an efficient solution.

Solution Approach

The implementation of the solution uses a **two-pointer technique** along with basic arithmetic calculations to find the number of homogenous substrings. Here is the walkthrough of the code:

- The function `countHomogenous` starts by initializing the variable `mod` to `10**9 + 7` for modulo operations to prevent overflow.
- Two variables are declared, `i` being the start pointer (initialized at index 0) and `n` being the length of the input string `s`.
- We also initialize a variable `ans` to store the cumulative number of homogenous substrings found.

The solution enters a loop that continues until the start pointer `i` has reached the end of the string (`i < n`):

1. A second pointer `j` is set to start at the same position as `i`. This will be used to find the end of the current homogenous substring.
2. A while loop is used to move `j` forward as long as `s[j]` is the same as `s[i]`. When `s[j]` is different from `s[i]`, it means we have found the end of the homogenous substring.
3. After the while loop, we now have a substring from index `i` to `j - 1` that is homogenous. The length of this substring is `cnt = j - i`.
4. To find the number of homogenous substrings within this section, we use the arithmetic series sum formula `(1 + cnt) * cnt / 2`, where `cnt` is the length of the homogenous substring.
5. The result is added to `ans`, which keeps the running total of homogenous substrings. Every time a new count is added, we perform a modulo operation to make sure `ans` doesn't exceed `10**9 + 7`.
6. Finally, we move the start pointer `i` to the position where `j` ended, as everything before `j` is already part of a homogenous substring we've counted.

The use of the two-pointer technique efficiently reduces the time complexity since each character in the string is checked only once. By only considering stretches of identical characters and using the arithmetic series sum formula, we avoid having to individually count each possible substring. This is what makes the algorithm efficient.

The function ends by returning the total count `ans` as the resulting number of homogenous substrings.

Example Walkthrough

Let's illustrate the solution approach with a small example. Assume the input string `s = "abbba"`.

- The `countHomogenous` function begins by initializing the variables (`mod = 10^9 + 7`, `i = 0`, `n = 5`, and `ans = 0`).
- The main loop starts with `i < n`, which is true at the start (`i = 0`).

Starting with `i = 0`:

1. We set the second pointer `j = i = 0`. We are now looking for a homogenous substring starting at 'a'.
2. As we increment `j`, we realize that `s[j] == s[i]` only for `j = 0`. As soon as `j` becomes 1, `s[j]` becomes 'b', which is different from `s[i]` ('a'). So the while loop stops with `j = 1`.
3. We found a homogenous substring 'a' from index `0` to `0`. Thus `cnt = j - i = 1 - 0 = 1`.
4. The number of homogenous substrings for this `cnt` is `(1 + 1) * 1 / 2 = 1`.
5. We add this to `ans`, `ans = 0 + 1 = 1`.
6. We then set `i = j`, so `i` is now `1` and we start looking for a new homogenous substring.

Next with `i = 1`:

1. Set `j` to `1`. The character at this index is 'b'.
2. We increment `j` while `s[j]` is the same as `s[i]`. This gives us `j = 4` because indices 1, 2, and 3 are all 'b'.
3. We found a homogenous substring 'bbb' from index `1` to `3`, so `cnt = j - i = 4 - 1 = 3`.
4. Using the formula for `cnt = 3`, the number of homogenous substrings is `(1 + 3) * 3 / 2 = 6`.
5. This is added to `ans`, which becomes `ans = 1 + 6 = 7`.
6. The `i` is moved to the index `4`.

The last character is evaluated but as there are no repeating characters beyond this point, `j` never increments. Therefore, this results in a `cnt` of `1`, which provides a single homogenous substring 'a'.

The answer, or total number of homogenous substrings, is summed up as `ans = 7 + 1 = 8`, which is the result returned by the function.

This walkthrough simplifies how the solution makes efficient use of the two-pointer technique to calculate the number of homogenous substrings without having to count each one individually.

Python Solution

```python
class Solution:
    def countHomogenous(self, s: str) -> int:
        # Define the modulus as mentioned in the problem statement.
        modulo_factor = 10**9 + 7

        # Initialize starting index for iterating the string and the length of the string.
        current_index, string_length = 0, len(s)

        # Initialize the answer to count homogenous substrings.
        homogenous_count = 0

        # Iterate over the string to find homogenous substrings.
        while current_index < string_length:
            # Find the end of the current homogenous substring.
            next_index = current_index
            while next_index < string_length and s[next_index] == s[current_index]:
                next_index += 1

            # Calculate the length of the current homogenous substring.
            substring_length = next_index - current_index

            # Count the number of homogenous substrings which can be formed from
            # the current homogenous substring using the formula for the sum of
            # the first n natural numbers: n * (n + 1) / 2.
            homogenous_count += (1 + substring_length) * substring_length // 2

            # Use modulo operation to avoid large numbers as per problem constraints.
            homogenous_count %= modulo_factor

            # Move to the beginning of the next potential homogenous substring.
            current_index = next_index

        # Return the final count of homogenous substrings.
        return homogenous_count

```

Java Solution

```java
class Solution {
    private static final int MOD = (int) 1e9 + 7;

    public int countHomogenous(String s) {
        // Length of the input string
        int length = s.length();
        // Variable to hold the total count of homogenous substrings
        long totalHomogenousSubstrings = 0;

        // Loop through the string characters
        for (int startIndex = 0, endIndex = 0; startIndex < length; startIndex = endIndex) {
            // Set the end index to the current start index
            endIndex = startIndex;
            // Extend the end index while the end character is the same as the start character
            while (endIndex < length && s.charAt(endIndex) == s.charAt(startIndex)) {
                endIndex++;
            }
            // Calculate the length of the homogeneous substring
            int homogeneousLength = endIndex - startIndex;
            // Use the formula for sum of first n natural numbers to calculate the number of substrings
            totalHomogenousSubstrings += (long) (1 + homogeneousLength) * homogeneousLength / 2;
            // Apply modulo operation to prevent overflow
            totalHomogenousSubstrings %= MOD;
        }
        // Cast the result to int before returning, since the final output must be an integer
        return (int) totalHomogenousSubstrings;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    static constexpr int MOD = 1e9 + 7; // Define the modulus constant for preventing integer overflow.

    // Method to count the number of homogenous substrings in the given string s.
    int countHomogenous(string s) {
        int length = s.size(); // The length of the input string.
        long long answer = 0;  // To store the final answer, initialized to 0.

        // Loop through the string to count all homogenous substrings.
        for (int start = 0, end = 0; start < length; start = end) {
            // Find the end of the current homogenous substring.
            end = start;
            while (end < length && s[end] == s[start]) {
                ++end;
            }

            // Compute the count of characters in the current homogenous substring.
            int count = end - start;

            // Calculate the number of possible homogenous substrings,
            // which is the sum of the first count natural numbers: count * (count + 1) / 2.
            // We use long long to avoid integer overflow during the calculation.
            answer += static_cast<long long>(count + 1) * count / 2;
            answer %= MOD; // Apply the modulus to keep the answer within the integer limits.
        }

        return static_cast<int>(answer); // Return the answer as an integer.
    }
};

```

Typescript Solution

```typescript
// Function to count the number of homogenous substrings in a given string.
// A homogenous substring is one that consists of a single unique character.
// For example, in the string "aa", there would be three homogenous substrings: "a", "a", and "aa".
function countHomogenous(s: string): number {
    const MODULO: number = 1e9 + 7; // Define a constant for modulo to avoid large numbers
    const n: number = s.length; // Length of the input string
    let count: number = 0; // Initialize the count of homogenous substrings

    // Use two pointers to iterate through the string
    for (let startIndex = 0, currentIndex = 0; currentIndex < n; currentIndex++) {
        // If the current character is different from the starting character,
        // update the starting index to the current index
        if (s[startIndex] !== s[currentIndex]) {
            startIndex = currentIndex;
        }
        // Calculate the number of homogenous substrings found so far based on the current sequence
        // Add the number of new homogenous substrings from startIndex to currentIndex
        count = (count + (currentIndex - startIndex + 1)) % MODULO;
    }
    // Return the total count of homogenous substrings modulo the defined constant
    return count;
}

```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the string `s`. This is because each character in the string is checked exactly once to form homogenous substrings (characters that are the same and contiguous). The inner `while` loop runs only once for each homogenous substring, and since it only moves `j` to the end of a homogenous substring, the iterations of the inner loop throughout the entire run of the algorithm sum up to `O(n)`.

The space complexity of the code is `O(1)`. This is because the algorithm uses a constant number of additional variables (`mod`, `i`, `n`, `ans`, `j`, `cnt`) which do not scale with the input size - they use a constant amount of space regardless of the length of the string `s`.