2262. Total Appeal of A String

Hard | Hash Table | String | Dynamic Programming | Leetcode Link

Problem Description

The goal is to calculate the total appeal of all possible substrings of a given string. The appeal of a string is determined by the number of distinct characters it contains. For instance, the string "abbca" contains three distinct characters, 'a', 'b', and 'c', therefore its appeal is 3.

To find the total appeal, we must consider each substring of the input string. A substring is defined as any contiguous sequence of characters within the string. For example, the string "abc" has the following substrings: "a", "b", "c", "ab", "bc", "abc". Each substring has its own appeal, and the total appeal is the sum of the appeals of all these substrings.

The challenge lies in efficiently computing this sum without having to explicitly check each substring, as there could be a large number of them.

Intuition

The presented solution takes a dynamic approach to calculate the total appeal. It optimizes the process by not generating each substring, but instead, keeping track of how each new character added to the end of the substrings affects their overall appeal.

The crux of the approach relies on two key observations:

1. When a new character is added that hasn't been seen before in any of the previous substrings, it increases the appeal of all the substrings ending at the previous character by one.
2. If the new character has been seen before, only the substrings that started after the last occurrence of this character have their appeal increased by one.

To keep track of these effects, an array `pos` is used to record the last position where each character occurred in the string (`-1` if the character hasn't occurred yet).

As we iterate through the string character by character, we adjust the running total appeal `t` by considering the difference between the current index `i` and the last occurrence `pos[c]`. This efficiently calculates the incremental appeal contributed by the current character to all substrings ending with it.

By summing up these incremental contributions `t` as we go along, we obtain the total appeal of all substrings by the end of the traversal. This method avoids the overhead of explicit substring enumeration and appeal calculation, leading to a much faster solution.

Solution Approach

The solution implements an efficient algorithm to calculate the total appeal of all substrings of a given string. It uses a simple linear-time algorithm that leverages a concept similar to dynamic programming, along with an integer array to keep track of the last occurrence of characters.

The key steps in the algorithm include:

1. Initialize an array `pos` with length 26 (since there are 26 lowercase letters in the English alphabet) and fill it with `-1`. This array is used to store the last position where each character was seen in the string.

2. Initialize two integers: `ans` to accumulate the total appeal and `t` to keep track of the current sum of appeals as the algorithm iterates through the string.

3. Iterate over each character `c` in the string `s`, using an index `i` to keep track of the position.

4. Convert the character `c` into an array index (0-25) by subtracting the ASCII code of 'a' from the ASCII code of `c`.

5. Update `t` by adding the difference between `i` and the last occurrence of `c` (which is `pos[c]`). This captures the increase in appeal for all substrings ending at the current character, as explained in the intuition section.

6. Add the updated `t` to `ans`, incrementally building up the total appeal.

7. Finally, update the last occurrence of `c` in `pos` to the current index `i`.

8. Continue the iteration until the end of the string and return `ans` as the total appeal.

The code uses the fact that updating the total appeal with each new character and calculating the incremental appeal based on the last occurrence is sufficient to count the appeal for all possible substrings.

This approach results in `O(n)` time complexity, where `n` is the length of the input string. It avoids the need for nested loops, which would result in a higher time complexity. Additionally, the space complexity is `O(1)` since the auxiliary space used (the `pos` array) does not grow with the size of the input string.

Example Walkthrough

Let's walk through an example to illustrate the solution approach using the string `"abaca"`.

1. **Initialization**: We start by initializing the array `pos` with size 26, to represent each letter in the English alphabet, and fill it with `-1`, indicating that none of the characters have been seen yet. We also initialize `ans` (accumulated total appeal) and `t` (current sum of appeals) to 0.

   `pos` is initially `[-1, -1, -1, ..., -1]`, `ans = 0`, and `t = 0`.

2. **Iteration**:
   - **Character 'a' at index 0**: The index for 'a' is `0 - 'a' = 0`. Since 'a' was not seen before (`pos[0] == -1`), `t` is updated by `0 - (-1) = 1`. Then `ans` is updated by adding `t` to it (now `ans = 1`). The position of 'a' is updated in `pos` as `pos[0] = 0`.
   - **Character 'b' at index 1**: Similar to 'a', 'b' has not been seen before. The index for 'b' is `1 - 'a' = 1`. `t` is updated by `1 - (-1) = 2`. Now `ans += t` (now `ans = 3`). Update `pos[1]` with the current index (`pos[1] = 1`).
   - **Character 'a' at index 2**: 'a' has been seen before at index 0. The index for 'a' is `0`. `t` is updated by `2 - 0 = 2`. Add `t` to `ans` (now `ans = 3 + 2 = 5`). Update `pos[0]` with the current index (`pos[0] = 2`).
   - **Character 'c' at index 3**: 'c' has not been seen before. The index for 'c' is `2 - 'a' = 2`. `t` is updated by `3 - (-1) = 4`. `ans += t` (now `ans = 5 + 4 = 9`). Update `pos[2]` with the current index (`pos[2] = 3`).
   - **Character 'a' at index 4**: 'a' has been seen before at index 2. The index for 'a' is `0`. `t` is updated by `4 - 2 = 2`. `ans += t` (now `ans = 9 + 2 = 11`). Update `pos[0]` with the current index (`pos[0] = 4`).

3. **Result**: After iterating through the entire string, `ans` holds the total appeal of all possible substrings, which is `11` for the example string `"abaca"`.

This walkthrough demonstrates how the solution efficiently calculates each character's contribution to the total appeal as we process the string, resulting in an optimal calculation without examining each substring individually.

Python Solution

```python
class Solution:
    def appealSum(self, s: str) -> int:
        total_appeal = current_sum = 0  # Initialize total appeal and current sum of appeal
        last_positions = [-1] * 26  # Initialize list to store the last positions of characters

        # Loop through the string with index and character
        for index, char in enumerate(s):
            char_code = ord(char) - ord('a')  # Convert character to a number (0-25)

            # Update current sum by adding the difference between current index and last seen position
            current_sum += index - last_positions[char_code]

            # Add the updated current sum to the total appeal
            total_appeal += current_sum

            # Update the last seen position of this character
            last_positions[char_code] = index

        return total_appeal  # Return the total appeal of the substring

```

Java Solution

```java
class Solution {
    // This method calculates the sum of the appeal of all substrings of a given string s.
    // The appeal of a string is defined as the number of distinct characters found in the string.
    public long appealSum(String s) {
        long totalAppeal = 0;   // This variable will store the sum of the appeal of all substrings.
        long currentAppeal = 0; // This variable stores the appeal of the substring ending at the current character.
        int[] lastPosition = new int[26]; // An array to store the last position of each character.
        Arrays.fill(lastPosition, -1); // Initialize last positions to -1 for all characters.

        // Iterate over each character in the string to compute the appeal of all possible substrings.
        for (int i = 0; i < s.length(); ++i) {
            int charIndex = s.charAt(i) - 'a'; // Convert the char to an index 0-25 corresponding to 'a'-'z'.
            // Update the current appeal by adding the contribution of the current character.
            // The contribution is the difference between the current position and its last seen position.
            currentAppeal += i - lastPosition[charIndex];
            // Add the current appeal to the total appeal.
            totalAppeal += currentAppeal;
            // Update the last seen position for the current character.
            lastPosition[charIndex] = i;
        }

        // Return the total appeal sum of all substrings.
        return totalAppeal;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    long long appealSum(string s) {
        long long totalAppeal = 0; // This will hold the sum of appeals of all substrings.
        long long currentAppeal = 0; // This will keep track of the appeal of the current substring.
        vector<int> lastPosition(26, -1); // Keeps track of the last position of each character in the alphabet within the string.

        // Loop through each character in the string.
        for (int i = 0; i < s.size(); ++i) {
            int charIndex = s[i] - 'a'; // Convert the current character to an index (0-25) corresponding to a-z.

            // The appeal of a substring extends by the distance from the last occurrence of the current character.
            currentAppeal += i - lastPosition[charIndex];

            // Add the current substring's appeal to the total.
            totalAppeal += currentAppeal;

            // Update the last seen position for the current character.
            lastPosition[charIndex] = i;
        }

        // Return the total appeal which is the sum of appeals of all substrings.
        return totalAppeal;
    }
};

```

Typescript Solution

```typescript
function appealSum(s: string): number {
    // Initialize an array to keep track of the last seen position of each character
    // and fill it with -1, indicating that none have been seen yet.
    const lastPosition: number[] = Array(26).fill(-1); // 26 letters in the alphabet
    const stringLength = s.length;
    let totalAppeal = 0; // This will accumulate the total appeal of all substrings
    let currentAppeal = 0; // This keeps the running sum of appeals

    // Iterate through each character in the string
    for (let index = 0; index < stringLength; ++index) {
        // Convert the current character to its alphabet position (0 for 'a', 1 for 'b', etc.)
        const charCode = s.charCodeAt(index) - 'a'.charCodeAt(0);
        // Update the running sum of appeals by adding the distance from the last seen position
        currentAppeal += index - lastPosition[charCode];
        // Add the current appeal to total appeal
        totalAppeal += currentAppeal;
        // Update the last seen position of the current character
        lastPosition[charCode] = index;
    }

    // Return the calculated total appeal
    return totalAppeal;
}

```

Time and Space Complexity

The given code calculates the sum of the appeals of all substrings of a string.

Time Complexity

We iterate through each character in the input string only once. The loop runs for `n` iterations if `n` is the length of the input string `s`.

Inside the loop, we perform constant-time operations: indexing an array (`pos[c]`), arithmetic operations (`t += i - pos[c]` and `ans += t`), and updating an array element (`pos[c] = i`). Since these operations do not depend on the size of `n` and are done in constant time, the time complexity of the loop is `O(n)`.

Therefore, the overall time complexity of the code is `O(n)`.

Space Complexity

We are using an extra array `pos` of size `26` to keep track of the last positions of each character that appears in the string. The size of this array depends on the size of the character set `|\Sigma|`, which in this case is the English alphabet and hence `|\Sigma| = 26`.

Therefore, the space complexity of the code is `O(|\Sigma|)`, which is `O(26)` for this problem.

Since `26` is a constant and does not change with the input size, we could also consider the space complexity as `O(1)`, constant space.