2083. Substrings That Begin and End With the Same Letter

Medium Hash Table Math String Counting Prefix Sum

Problem Description

The problem presents a situation where we are given a string `s`, which contains only lowercase English letters. Our goal is to determine the total number of substrings within this string that start and end with the same character. It's essential to remember that a substring is simply a sequence of characters that are adjacent in the string and that the string in question uses 0-based indexing.

For example, if given the string `s = "ababa"`, some valid substrings that begin and end with the same character would be `["a", "aba", "ababa", "b", "bab", "a"]`. In this challenge, we need to enumerate all possible substrings that meet this criterion and return their count.

Intuition

The solution strategy revolves around the observation that for each occurrence of a character in the string, it can potentially form a substring with each of its previous appearances. This is because any sequence of characters between the two same characters (including no characters at all) will still result in a substring that starts and ends with that character.

For example, if we have the string `s = "aa"`, then there are three such substrings: `"a"`, `"a"`, `"aa"`. For the first 'a', there is 1 substring which is itself. For the second 'a', there are 2 substrings: itself and the substring "aa" that includes both the first and the second 'a'.

To implement this, we utilize a dictionary to keep track of the count of each character as we iterate through the string. Whenever we encounter a character, we increment its count in the dictionary. The current count of the character gives us the number of substrings that end with this character and start with a previous instance of the same character. We add this count to our answer for each character we encounter. The sum of all these counts will give us the total number of desired substrings.

The `Counter` class from the `collections` module in Python provides a convenient way to count occurrences of each character. The variable `cnt` is our counter, and `ans` is used to accumulate the count of valid substrings. We iterate through each character in the string, update its count in `cnt`, and add the updated count to `ans`, yielding the total number of substrings that start and end with the same character by the time we've processed the entire string.

Solution Approach

The implementation uses a hash table to keep track of how many times each character has appeared in the string so far. The hash table is implemented using the `Counter` class from Python's `collections` module, which efficiently handles the counting of hashable objects.

Here's a step-by-step breakdown of the implementation:

1. A `Counter` object named `cnt` is created to keep track of the occurrences of each character in the string.
2. An integer variable `ans` is initialized to `0`. It will serve as an accumulator for the total count of substrings satisfying the given condition.
3. The code enters a loop that iterates over each character `c` in the string `s`.
4. For each character, the loop increments its count in `cnt`. This is achieved by the expression `cnt[c] += 1`. In the `Counter`, if the character `c` doesn't exist yet, it would be initialized to `0` before being incremented. So, after this step, `cnt[c]` contains the number of times `c` has appeared so far.
5. The current value of `cnt[c]` is then added to `ans`. This is based on the observation that if we have seen a character `c` times so far, there are `n` substrings that can end with this instance of the character, each starting with one of the previously seen instances of this character (including a substring just consisting of this one character).
6. After the loop ends, `ans` contains the total number of qualified substrings, and the function returns this value.

This approach is effective for the problem at hand because it leverages the fact that the number of substrings ending with a certain character equates to the instances of that character seen so far. This allows the algorithm to count the requisite substrings in a single linear pass over the string, thus having a time complexity of O(n), where n is the length of the string.

Here is the reference solution approach implemented in Python:

```
class Solution:
    def numberOfSubstrings(self, s: str) -> int:
        cnt = Counter()
        ans = 0
        for c in s:
            cnt[c] += 1
            ans += cnt[c]
        return ans
```

In this code, `for c in s` iterates over the characters in the string. The `Counter` object `cnt` keeps track of the occurrence count, and the accumulator `ans` aggregates the number of substrings. After visiting all characters, the total count is held in `ans`, which is returned as the final result.

Example Walkthrough

Let's use a small example to illustrate the solution approach described above. Suppose we have the string `s = "abcab"`.

1. We start with an empty `Counter` object `cnt` and an accumulator `ans`, initialized to `0`.
2. We iterate through each character in the string, in order.
3. Upon encountering the first character `a`, we update the counter `cnt['a'] += 1`. Now, `cnt = Counter({'a': 1})`, and we increment `ans` by `cnt['a']`, so `ans = 1`.
4. Next, we see the character `b`. We update `cnt['b'] += 1`, making `cnt = Counter({'a': 1, 'b': 1})`, and append `cnt['b']` to `ans`, resulting in `ans = 2`.
5. The third character is `c`. We update `cnt['c'] += 1`, giving us `cnt = Counter({'a': 1, 'b': 1, 'c': 1})`, and increment `ans` by `cnt['c']`, leading to `ans = 3`.
6. The fourth character is `a` again. This time, `cnt['a'] += 1` makes `cnt = Counter({'a': 2, 'b': 1, 'c': 1})` because `a` has appeared twice. We add `cnt['a']` to `ans`, and now `ans = 5` (since `cnt['a']` is now `2`, we add `2`).
7. Finally, we see another `b`. We increment `cnt['b']`, so `cnt = Counter({'a': 2, 'b': 2, 'c': 1})`, and then `ans += cnt['b']` makes `ans = 7`.

At the end of the loop, we have considered every character and updated our accumulator `ans` based on the occurrences of each character. The final value of `ans` is `7`, which is the number of substrings that start and end with the same character in the string `s = "abcab"`.

These substrings are specifically `"a"`, `"b"`, `"c"`, `"abca"`, `"bcab"`, `"aba"`, and `"bab"`. Notice how substrings that start and end with the same character and contain other characters in the middle are also included, as illustrated by `"abca"` and `"bcab"`.

Solution Implementation

Python

```
from collections import Counter

class Solution:
    def numberOfSubstrings(self, strina: str) -> int:
        # Initialize an empty Counter to keep track of character frequencies.
        char_count = Counter()

        # Initialize the answer variable to store the result.
        total_substrings = 0

        # Iterate over each character in the string.
        for char in strina:
            # Increment the count of the current character.
            char_count[char] += 1

            # The number of substrings ending with the current character is equal
            # to its current count (since any substring ending before could be extended by this character).
            total_substrings += char_count[char]

        # Return the total number of substrings.
        return total_substrings
```

Java

```
class Solution {
    public long numberOfSubstrings(String s) {
        // Create an array to store the count of each letter.
        int[] letterCount = new int[26];

        // Initialize a variable to store the total number of substrings.
        long totalSubstrings = 0;

        // Iterate over each character in the string.
        for (int i = 0; i < s.length(); ++i) {
            // Calculate the index of the current character 'a' being 0, 'b' being 1, ... 'z' being 25.
            int index = s.charAt(i) - 'a';

            // Increment the count of the current letter in the letter count array.
            ++letterCount[index];

            // Each time a new character is processed, increment totalSubstrings by the count of that character.
            // The count represents the number of substrings ending with that character.
            totalSubstrings += letterCount[index];
        }

        // Return the total number of substrings.
        return totalSubstrings;
    }
}
```

C++

```
class Solution {
public:
    long long numberOfSubstrings(string s) {
        int count[26] = {}; // Initialize array to store the count of each letter
        long long answer = 0; // Initialize the answer variable to store the total number of substrings

        // Loop through each character in the string
        for (char& c : s) {
            // Increment the count of the current letter and add the new count to the answer
            // The new count reqresents the number of substrings ending with the current letter
            // that have not been counted before
            answer += ++count[c - 'a'];
        }

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

TypeScript

```
// Function to count the number of substrings
function numberOfSubstrings(s: string): number {
  // Initialize an array to store the count of each letter ('a' to 'z')
  let count: number[] = new Array(26).fill(0);
  // Initialize the answer variable to store the total number of substrings
  let answer: number = 0;

  // Loop through each character in the string
  for (const c of s) {
    // Increment the count of the current letter (using its ASCII value to map to an index)
    // The new count reqresents the number of substrings ending with the current letter
    // that have not been counted before
    answer += ++count[c.charCodeAt(0) - 'a'.charCodeAt(0)];
  }

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

// Note: Since this function is defined globally, it can be called directly with a string parameter.
```

```
from collections import Counter

class Solution:
    def numberOfSubstrings(self, strina: str) -> int:
        # Initialize an empty Counter to keep track of character frequencies.
        char_count = Counter()

        # Initialize the answer variable to store the result.
        total_substrings = 0

        # Iterate over each character in the string.
        for char in strina:
            # Increment the count of the current character.
            char_count[char] += 1

            # The number of substrings ending with the current character is equal
            # to its current count (since any substring ending before could be extended by this character).
            total_substrings += char_count[char]

        # Return the total number of substrings.
        return total_substrings
```

Time and Space Complexity

The given code calculates the number of substrings that can be formed in a string with each character being counted once it is part of a substring as it's introduced.

Time Complexity:

Let's analyze the time complexity of the provided function:

1. The function iterates through each character in the string `s` which has a length of `n`.
2. In each iteration, it performs an update on the `Counter()` dictionary and an addition operation to the `ans` variable.

The `Counter()` update operation and addition operation are both `O(1)` operations.

Hence, the loop runs `n` times with constant-time operations within it, resulting in a time complexity of `O(n)`.

Space Complexity:

Now, let's analyze the space complexity:

1. The `Counter()` data structure is used to keep a count of each character. In the worst case, this would store as many keys as there are unique characters in the string `s`.
2. Assuming the input string `s` could include all possible characters in the charset used (let's say ASCII for simplicity), there is a constant number of possible characters, which means the `Counter` would have at most `C` keys, where `C` is the size of the charset.

Therefore, since the size of the charset is fixed and does not grow with the input size `n`, the space required by the `Counter` is `O(1)`.

As a result, the space complexity of the code is `O(1)` if we consider the size of the charset to be constant and not dependent on the size of the input string. If considering unicode or a variable character set that indeed scales with `n`, it would be `O(n)` for space complexity.