2947. Count Beautiful Substrings I

Medium String Enumeration Prefix Sum

Problem Description

In this LeetCode problem, we are given a string `s` and an integer `k`. The string `s` consists of lowercase English letters. We consider a string beautiful if it obeys two conditions:

1. The count of vowels in the string equals the count of consonants. (Vowels are 'a', 'e', 'i', 'o', 'u'; all other letters are consonants).
2. The product of the number of vowels and consonants is divisible by `k`.

Our task is to find and return the number of non-empty substrings of `s` that are beautiful.

A substring is defined as a contiguous segment of the string, and each individual character is also considered a substring of itself.

Intuition

To solve this problem, we can employ a brute-force approach by checking every possible substring of the string `s` and determine whether it qualifies as beautiful according to the given criteria.

We initialize a count (`ans`) that will keep track of the number of beautiful substrings found.

Starting with the first character, we iterate through the string `s` using two nested loops: the outer loop sets the starting index of a substring, and the inner loop expands the substring by advancing the ending index.

For each possible substring, we maintain a vowel count. As we expand the substring by moving the ending index, we update the vowel count if the current character is a vowel.

Since we know the total length of the substring, we can find the number of consonants by subtracting the vowel count from the substring length.

We then check the first condition: whether the number of vowels equals the number of consonants. If this condition is not met, the substring cannot be beautiful and we continue with the next iteration.

If the first condition is satisfied, we compute the product of vowels and consonants and check if it is divisible by `k` to fulfill the second condition.

Each time both conditions are met, we increment our count (`ans`) as we've found a beautiful substring.

After checking all possible substrings, we return the count as our answer.

Here's a caveat: the above approach might not be optimal for strings with very large lengths, as the time complexity is O(n^2), which can cause timeouts on such inputs. However, the solution is quite straightforward and covers the basic brute-force approach to this problem.

Solution Approach

The implementation of the solution follows a brute-force approach in which we consider every possible non-empty substring of the input string `s` and check whether it is beautiful or not. Here is a walk-through of the algorithm:

- We initiate a `for` loop with the variable `i` that will serve as the starting index of the substring we're evaluating.
- For every `i`, we initiate an inner `for` loop with the variable `j` which represents the ending index of the current substring.
- A set `vs` contains all the vowel characters for constant time check of whether a character is a vowel or not.
- We maintain a variable `vowels` to keep a count of the vowels in the current substring (from `i` to `j`).
- Inside the inner loop, we check if the character at the current `j` index is a vowel by checking its membership in the `vs` set. If it is a vowel, we increment the `vowels` count.
- We then calculate the number of consonants in the current substring as the difference between the length of the substring (which is `j - i + 1`) and the number of vowels.
- If the count of vowels equals the count of consonants (`vowels == consonants`), we check the second condition, which is whether their product is divisible by `k` (`vowels * consonants % k == 0`). If both conditions are fulfilled, the substring is beautiful and we increment our result counter `ans`.
- We repeat this process for every `j` and `j` until all substrings have been considered.
- After the loops complete, the value of `ans` holds the total number of beautiful substrings and we return this value.

Let's examine the components of the code:

- **Variables**: `i` for the length of the input string, `vs` for the set of vowels, `ans` as the counter for beautiful substrings.
- **Data Structures**: We utilize a `set` data structure (`vs`) for fast lookup to check if a character is a vowel.
- **Loops**: We use nested `for` loops to go through each possible substring.

Here's the code snippet that encapsulates the above logic:

```
class Solution:
    def beautifulSubstrings(self, s: str, k: int) -> int:
        n = len(s)
        vs = set("aeiou")
        ans = 0
        for i in range(n):
            vowels = 0
            for j in range(i, n):
                vowels += s[j] in vs
                consonants = j - i + 1 - vowels
                if vowels == consonants and vowels * consonants % k == 0:
                    ans += 1
        return ans
```

The time complexity of this solution is O(n^2) because we examine each substring and the space complexity is O(1) since we only use a fixed amount of additional space.

Example Walkthrough

Let's walk through a small example to illustrate this solution. Suppose we have the string `s = "azecb"` and we need to check for beautiful substrings with `k = 2`.

1. We have the variable `i` ranging from 0 to 4 (for length of `s` which is 5). Let's examine the loops step by step when `i = 0`.
2. We also define `vs` as a set containing `'a', 'e', 'i', 'o', 'u'` to keep track of vowels.
3. Starting with `i = 0`, we set `vowels = 0`. This represents the starting index of our potential substring. Now, we will execute the inner loop with `j` ranging also from 0 to 4:
   - For `j = 0`:
     - Our substring is `"a"` which has 1 vowel.
     - Since there are no consonants, this substring isn't beautiful, we move to the next iteration.
   - For `j = 1`:
     - Our substring is `"az"`. Now we have 1 vowel and 1 consonant.
     - The number of vowels equals the number of consonants and their product is 1 which is not divisible by 2. Not beautiful, continue.
   - For `j = 2`:
     - Substring `"aze"`. We have 2 vowels, 1 consonant.
     - Vowel and consonant counts do not match, continue.
   - For `j = 3`:
     - Substring `"azec"`. 2 vowels, 2 consonants.
     - Counts match, and their product is 4 which is divisible by 2. Beautiful substring! Increment `ans` to 1.
   - For `j = 4`:
     - Substring `"azecb"`. 2 vowels, 3 consonants.
     - Counts do not match, continue loop.
4. Repeat the same process, moving `j` from 1 to 4, each time resetting `vowels` to 0, and checking each possible substring that begins at that `i`.

After the loops complete, we have found all possible beautiful substrings and we have `ans = 1` as our result, meaning there is 1 beautiful substring in "azecb" when `k = 2`.

Hence, the `beautifulSubstrings` function would return 1.

This example illustrated the process of using the brute-force approach in a smaller scenario. The actual function would perform similar steps for each index and count all such occurrences, giving us the total count of beautiful substrings.

Solution Implementation

Python

```
class Solution:
    def beautifulSubstrings(self, input_string: str, divisor: int) -> int:
        # Length of the input string.
        string_length = len(input_string)

        # Set containing vowels for quick lookup.
        vowel_set = {'a', 'e', 'i', 'o', 'u'}

        # Counter for the beautiful substrings.
        beautiful_count = 0

        # Loop through each character in the string as the starting point.
        for start_index in range(string_length):
            # Initialize the count of vowels found.
            vowel_count = 0

            # Explore all substrings that begin at start_index.
            for end_index in range(start_index, string_length):
                # Increment vowel count if the current character is a vowel.
                vowel_count += input_string[end_index] in vowel_set

                # Calculate number of consonants in the current substring.
                consonant_count = (end_index - start_index + 1) - vowel_count

                # Check if the substring is beautiful:
                # The number of vowels and consonants must be equal and
                # their product must be divisible by the divisor `k`.
                if vowel_count == consonant_count and (vowel_count * consonant_count) % divisor == 0:
                    beautiful_count += 1

        # Return the total count of beautiful substrings.
        return beautiful_count
```

Java

```
class Solution {
    public int beautifulSubstrings(String inputString, int divisor) {
        int stringLength = inputString.length(); // Store the length of the input string.

        // Initialize an array to mark vowels with '1' and consonants with '0'.
        int[] vowelStatus = new int[26];
        for (char vowel : "aeiou".toCharArray()) {
            vowelStatus[vowel - 'a'] = 1; // Marking vowels in the array.
        }

        int totalCount = 0; // Initialize the count of beautiful substrings.

        // Loop through each character of the string as a starting point.
        for (int i = 0; i < stringLength; ++i) {
            int vowelCount = 0; // Initialize the count of vowels encountered.

            // Now check each substring starting from the current character.
            for (int j = i; j < stringLength; ++j) {
                // Increment the vowel count if a vowel is found.
                vowelCount += vowelStatus[inputString.charAt(j) - 'a'];

                // Calculate the number of consonants in the current substring.
                int consonantCount = j - i + 1 - vowelCount;

                // Check if the number of vowels and consonants are equal.
                if (vowelCount == consonantCount) {
                    // Calculate the product of vowel and consonant counts.
                    int product = vowelCount * consonantCount;
                    // If the product is divisible by the divisor, increment the total count.
                    if (product % divisor == 0) {
                        totalCount++;
                    }
                }
            }
        }

        // Return the total count of beautiful substrings.
        return totalCount;
    }
}
```

C++

```
class Solution {
public:
    int beautifulSubstrings(string str, int k) {
        int n = str.size();
        int vowelStatus[26] = {}; // Array to store the status of characters being vowel or not
        string vowels = "aeiou"; // String containing all vowels

        // Initialize the vowelStatus for vowels to 1
        for (char c : vowels) {
            vowelStatus[c - 'a'] = 1;
        }

        int answer = 0; // Variable to store the count of beautiful substrings

        // Iterate through the string
        for (int i = 0; i < n; ++i) {
            int vowelCount = 0; // Track the count of vowels in the substring
            // Explore all substrings starting at index i
            for (int j = i; j < n; ++j) {
                // Increment vowel count if current char is a vowel
                vowelCount += vowelStatus[str[j] - 'a'];
                // Get the count of consonants in the current substring
                int consonantCount = j - i + 1 - vowelCount;

                // Check if the substring is beautiful according to the given conditions
                if (vowelCount == consonantCount &&
                    (vowelCount * consonantCount) % k == 0) {
                    ++answer;
                }
            }
        }

        return answer; // Return the final count of beautiful substrings
    }
};
```

TypeScript

```
function beautifulSubstrings(s: string, k: number): number {
    const stringLength = s.length;
    // Create an array to keep track of vowels (1) and consonants (0)
    const vowelStatus = new Array(26).fill(0);
    // Identify and mark vowels in the vowelStatus array
    for (const char of 'aeiou') {
        vowelStatus[char.charCodeAt(0) - 'a'.charCodeAt(0)] = 1;
    }

    let beautifulCount = 0; // Counter for beautiful substrings
    // Iterate over the string
    for (let start = 0; start < stringLength; ++start) {
        let vowelCount = 0;
        // Explore all possible substrings starting from index 'start'
        for (let end = start; end < stringLength; ++end) {
            // Increment vowelCount if current character is a vowel
            vowelCount += vowelStatus[s.charCodeAt(end) - 'a'.charCodeAt(0)];
            // Calculate the number of consonants in the current substring
            const consonantCount = end - start + 1 - vowelCount;

            // Check if the current substring is 'beautiful'
            if (vowelCount === consonantCount && (vowelCount * consonantCount) % k === 0) {
                beautifulCount++;
            }
        }
    }

    return beautifulCount; // Return the total count of beautiful substrings
}
```

```
class Solution:
    def beautifulSubstrings(self, input_string: str, divisor: int) -> int:
        # Length of the input string.
        string_length = len(input_string)

        # Set containing vowels for quick lookup.
        vowel_set = {"a", "e", "i", "o", "u"}

        # Counter for the beautiful substrings.
        beautiful_count = 0

        # Loop through each character in the string as the starting point.
        for start_index in range(string_length):
            # Initialize the count of vowels found.
            vowel_count = 0

            # Explore all substrings that begin at start_index.
            for end_index in range(start_index, string_length):
                # Increment vowel count if the current character is a vowel.
                vowel_count += input_string[end_index] in vowel_set

                # Calculate number of consonants in the current substring.
                consonant_count = (end_index - start_index + 1) - vowel_count

                # Check if the substring is beautiful:
                # The number of vowels and consonants must be equal and
                # their product must be divisible by the divisor `k`.
                if vowel_count == consonant_count and (vowel_count * consonant_count) % divisor == 0:
                    beautiful_count += 1

        # Return the total count of beautiful substrings.
        return beautiful_count
```

Time and Space Complexity

Time Complexity

The given code has a nested loop structure where the outer loop runs 'n' times (where `n` is the length of string `s`) and the inner loop runs from `i` to `n`. On each iteration of the inner loop, it performs constant-time operations. The worst-case time complexity happens when all characters from `i` to `n` are executed, which is the sum of an arithmetic series.

The time complexity can be calculated as:

- The outer loop runs `n` times.
- The inner loop runs `n - i` times for every `i`.

So the total operations can be summed up as `n + (n-1) + (n-2) + ... + 1`. This is equivalent to `n * (n + 1) / 2`, which simplifies to `O(n^2)`.

Space Complexity

The space complexity of the code is determined by the extra space used which is independent of the input size `n`. Here, the variables used (`vowels`, `consonants`, `vs`, and `ans`) use constant space, and the set of vowels `vs` contains a fixed number of elements irrespective of `n`.

Thus, the space complexity is `O(1)`, which means it uses constant space.