2062. Count Vowel Substrings of a String

Easy Hash Table String

Leetcode Link

Problem Description

The problem is about identifying and counting special types of substrings within a given string. A substring is any consecutive sequence of characters from a string. What makes a substring special in this problem is that it is a "vowel substring," which means two things:

1. The substring consists only of vowels - which are the characters 'a', 'e', 'i', 'o', 'u'.
2. The substring contains all five of these vowels at least once.

Given a string `word`, the task is to return the number of such vowel substrings that exist within `word`.

For example, in the string "aeiouu", there is just one vowel substring that includes all five vowels, which is "aeiou". However, if you have extra vowels after like in "aeiouua", now there are two vowel substrings - "aeiou" and "aeiouu".

Intuition

Understanding the problem, the intuitive approach is to scan through the given string and extract all possible substrings, checking each one to determine if it qualifies as a vowel substring.

Here's a simple approach:

1. Create a set of vowels to use as a reference (`s`).
2. Iterate through all possible start positions (`i`) for a substring within the string `word`.
3. For each start position, iterate through all possible end positions (`j`) that create a substring.
4. For every start and end position, generate the substring `word[i:j]`.
5. Check if the set of characters in this substring matches the reference set of vowels exactly.
6. If it matches, it means the substring is a vowel substring.
7. Count all such instances to get the final answer.

The solution code uses a concise and elegant list comprehension combined with the `sum` function to count the number of times a generated substring's set of characters matches our reference set. This keeps the implementation succinct and avoids the need for verbose loops and conditionals.

Solution Approach

The Python solution provided uses a straightforward brute-force approach coupled with some clever use of the language's built-in data structures and functions. Here's more detail on how the algorithm and data structures are used:

1. **Set Data Structure**: The solution starts by creating a set `s` of all vowels. Sets are chosen because they allow O(1) average time complexity for checking if an element is contained within the set, which is useful in our case when we want to quickly check if a character is a vowel.

2. **Iteration**: The solution then iterates through all possible start and end positions for substrings within `word`. This is done using two nested loops. The outer loop (`for i in range(n)`) iterates from the start of the `word` string to its end. The inner loop (`for j in range(i + 1, n + 1)`) iterates from the current start position to the end of the string, ensuring every possible non-empty substring is considered.

3. **List Comprehension and Summation**: For each pair of start and end positions found in these iterations, a substring is created with `word[i:j]`. This substring is then immediately turned into a set, which removes duplicates and allows for easy comparison with the set `s` of vowels. If the sets match, that means the substring contains all the vowels and only vowels, making it a valid vowel substring. The list comprehension does this for each possible substring, generating a list of true (1) and false (0) values that represent whether each substring meets the criteria.

4. **Counting Matches**: The `sum` function then adds up the list of 1s and 0s, effectively counting the number of valid vowel substrings in the original `word`.

This process leads to a computational complexity of O(n^3), where n is the length of the string, since we have two nested loops (O(n^2)) and set creation and comparison operations that can be up to O(n). Despite the seemingly high complexity, for typical inputs that LeetCode problems expect, this solution performs adequately well.

The actual implementation in the code is concise due to the use of a one-liner list comprehension, which is a powerful feature in Python that helps to write compact and readable code.

Here is the critical part of the code explaining the implementation:

```
s = set('aeiou')
return sum(set(word[i:j]) == s for i in range(n) for j in range(i + 1, n + 1))
```

Essentially, we're checking for each substring of `word` defined by `[i:j]` whether the set of unique characters matches our set of vowels `s`. We sum the total number of times this is `True` to find our answer.

Example Walkthrough

Let's take the string `"aeiouuaeei"` as an example – we'll walk through the solution approach to see how it works in finding the number of special vowel substrings.

We start by initializing a set `s` with the vowels `'a'`, `'e'`, `'i'`, `'o'`, and `'u'`. As we know, the string length (n) is 10 in this case.

Now, we need to check all possible substrings and determine if they're special vowel substrings. We do this by:

1. Iterating through `i` (start index of the substring) from 0 to 9.
2. For each `i`, we iterate `j` (end index of the substring) from `i+1` to 10.
3. We create a substring using `word[i:j]` and convert it into a set to remove duplicates.
4. We then compare this set to our initial vowels set `s`.
5. If the sets are equal, we've found a special vowel substring.

Let's walk through the first few iterations:

- **Iteration 1:** `i = 0`, `j = 5`, substring is `"aeiou"` which yields a set of `{'a', 'e', 'i', 'o', 'u'}`. This does not match `s`, so we don't count it.
- **Iteration 2:** `i = 0`, `j = 6`, substring is `"aeiouu"` which yields a set of `{'a', 'e', 'i', 'o', 'u'}`. This matches `s`, so we count one occurrence.
- ...
- We continue this process, checking all the possible substrings.

After we finish iterating over all the start and end indices, we sum up the true instances where the condition was met, and this gives us the number of special vowel substrings within the word. For `"aeiouuaeei"`, the valid substrings include `"aeiouu"`, `"aeiouua"`, `"aeiouuae"`, `"aeiouuaee"`, and `"aeiouuaeei"` - each containing all vowels at least once.

Using the provided code snippet with our example string:

```
s = set('aeiou')
n = len("aeiouuaeei")
return sum(set(word[i:j]) == s for i in range(n) for j in range(i + 1, n + 1))
```

The code will check all substrings and find that there are 5 substrings that match the criteria, returning `5` as the output. This approach, albeit brute force and suboptimal for very large strings, works efficiently for the input sizes usually encountered on LeetCode.

Python Solution

```
class Solution:
    def countVowelSubstrings(self, word: str) -> int:
        # Length of the input word
        length = len(word)

        # Set containing all vowels for comparison
        vowels_set = {'a', 'e', 'i', 'o', 'u'}

        # Using list comprehension and sum to count all substrings
        # that contain exactly the vowels 'a', 'e', 'i', 'o', 'u'
        # For every possible substring defined by indices i and j,
        # we check if the set of characters in that substring matches
        # the set of vowels. We sum up all such occurrences.
        return sum(set(word[i:j]) == vowels_set for i in range(length) for j in range(i + 1, length + 1))
```

Java Solution

```
class Solution {
    // Method to count the number of vowel substrings with all 5 English vowels
    public int countVowelSubstrings(String word) {
        int stringLength = word.length(); // Get the length of the word
        int count = 0; // Initialize the count of substrings

        // Iterate over each character in the word
        for (int i = 0; i < stringLength; ++i) {
            // Create a set to store unique vowels of the current substring
            Set<Character> uniqueVowels = new HashSet<>();

            // Start from the current character and check subsequent characters
            for (int j = i; j < stringLength; ++j) {
                char currentChar = word.charAt(j); // Get the current character
                // If the character is not a vowel, break the inner loop
                if (!isVowel(currentChar)) {
                    break;
                }
                // Add the vowel to the set of unique vowels
                uniqueVowels.add(currentChar);
                // If the set contains all 5 unique vowels, increment the count
                if (uniqueVowels.size() == 5) {
                    count++;
                }
            }
        }
        return count; // Return the total count of vowel substrings
    }

    // Helper method to determine whether a character is a vowel
    private boolean isVowel(char character) {
        // Compare the character to all vowels and return true if it matches any
        return character == 'a' || character == 'e' || character == 'i' || character == 'o' || character == 'u';
    }
}
```

C++ Solution

```
#include <unordered_set>
#include <string>

class Solution {
public:
    // Counts the number of substrings that contain all vowel characters (aeiou) at least once
    int countVowelSubstrings(string word) {
        int count = 0; // Initialize the count of vowel-only substrings
        int wordSize = word.size(); // Store the size of the word for the loop condition

        // Iterate through each character of the string as a starting point
        for (int start = 0; start < wordSize; ++start) {
            std::unordered_set<char> vowels; // Initialize a set to keep track of vowels in the current substring

            // Extend the current substring by considering each subsequent character
            for (int end = start; end < wordSize; ++end) {
                char currentChar = word[end];

                // If the current character is not a vowel, break out of the inner loop
                if (!isVowel(currentChar)) break;

                // Add the current vowel to the set
                vowels.insert(currentChar);

                // If we have encountered all 5 vowels, increment the count
                count += vowels.size() == 5;
            }
        }

        return count; // Return the total count of vowel-only substrings
    }

    // Helper function to determine if a character is a vowel
    bool isVowel(char c) {
        return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u';
    }
};

// Example usage:
// Solution solution;
// int result = solution.countVowelSubstrings("aeeiou"); // This would return the count of valid substrings
```

Typescript Solution

```
// Function to count vowel beauty substrings in a given word.
function countVowelSubstrings(word: string): number {
    let count = 0; // Initialize counter for beautiful substrings
    const length = word.length; // Store length of the word

    // Iterate through each character of the word
    for (let i = 0; i < length; ++i) {
        const vowels = new Set<string>(); // A set to store unique vowels encountered

        // Explore every substring starting at position 'i'
        for (let j = i; j < length; ++j) {
            const character = word[j]; // The current character

            // Check if character is a vowel
            if (character === 'a' || character === 'e' || character === 'i' || character === 'o' || character === 'u') {
                vowels.add(character); // Add the vowel to the set

                // Check if all 5 vowels have been encountered
                if (vowels.size === 5) {
                    count++; // Increment counter as all vowels are present
                }
            } else {
                // If it's not a vowel, break the loop as the current substring cannot be beautiful.
                break;
            }
        }
    }

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

Time and Space Complexity

The time complexity of the provided code is `O(n^3)`. This is because the code uses two nested loops iterating over the length of the word, resulting in `n^2` iterations, and within each iteration, it creates a substring and converts it to a set. The creation of a set from the substring takes `O(k)`, where `k` is the length of the substring, which averages to `n/2` over all possible substrings. This gives us a total of `O(n^2 * n/2)`, which simplifies to `O(n^3)`.

The space complexity is `O(n)`, primarily because of the storage required for the set of vowels `s` and the substring set in each iteration. The size of set `s` is constant (with 5 vowels), but the space used for the substring can grow up to the size of the input word in the worst case, hence the space complexity `O(n)`.