1839. Longest Substring Of All Vowels in Order

Medium String Sliding Window Leetcode Link

Problem Description

A string is considered to be **beautiful** if it adheres to two key conditions:

1. All five of the English vowels (`'a'`, `'e'`, `'i'`, `'o'`, `'u'`) must be present at least once within the string.
2. The characters in the string must be sorted in alphabetical order. This means all occurrences of `'a'` come before any `'e'`, and so on with the remaining vowels, ordered as `aeiou`.

Some examples to illustrate:

- The string `"aeiou"` qualifies as beautiful because it contains all five vowels in the correct order.
- `"aaaaeeeiiiioou"` is also beautiful as it also respects the vowel presence and order, despite repetitions.
- However, strings like `"uaeio"`, `"aeiiou"`, or `"aaaeeeooo"` do not meet the criteria and are not considered beautiful, either due to incorrect order or absence of certain vowels.

The task is to determine the length of the longest beautiful substring in a given string `word`, which is composed solely of English vowels. A substring is a consecutive sequence of characters taken from the string. If no beautiful substring exists, the answer should be `0`.

Intuition

To find the solution to this problem, we take a step-by-step approach by breaking down the string into distinguishable parts according to the character transitions. Our goal is to identify consecutive groups of the same vowel and note down their position and length. This way, we can later check if these groups form a valid sequence that matches our conditions for a beautiful string.

Here's how we can conceptualize our approach:

1. Traverse the given string while keeping track of sequences of identical characters. For instance, in the string `"aaeeiiooou"`, we'd identify the sequences as `['aa', 'ee', 'ii', 'oo', 'u']`.
2. Store information about these sequences in a way that we can later check the sequence order. In the implementation, this is done by storing pairs of the character and its sequence length in an array.
3. With this array of vowel sequences, we can now look for subsequences of five elements where each character is exactly one of the vowels in the correct order `'a'`, `'e'`, `'i'`, `'o'`, `'u'`. When we find such a sequence, we calculate its total length by summing the lengths of its constituents.
4. As we may have several eligible beautiful subsequences, we want to find the longest one. Therefore, we iterate through all possible subsequences that could be beautiful and retain the maximum length found.

By simplifying the problem to identifying and evaluating sequences of vowels, and checking for the longest valid sequence, we can effectively solve the problem in a straightforward and efficient manner.

Solution Approach

The implemented solution follows these steps:

1. **Initialize a list to store character sequence information**: The list `arr` is used to store tuples of characters and their sequence lengths. It starts empty and is populated as we traverse through the input string.
2. **Loop through the string to fill the sequence list**: The outer `while` loop keeps track of our position `i` in the string `word`. For each position `i`, an inner `while` loop counts the length of the sequence of the same character starting from that position. The character and its sequence length are then appended as a tuple to the `arr` list. The value of `i` is updated to the position following the end of the current sequence.
3. **Initialize a variable to keep track of the answer**: `ans` is initialized to 0 and is used to record the length of the longest beautiful substring. It will be updated throughout the algorithm whenever a longer beautiful substring is found.
4. **Search for beautiful substrings**: A `for` loop iterates through the `arr` list, checking combinations of 5 consecutive character sequences. It extracts these five sequences using slicing (e.g., `b, c, d, e = arr[i : i + 5]`) and checks if the characters form the sequence `'aeiou'`.
5. **Update the maximum length if a beautiful substring is found**: If the sequence of characters is correct, it computes the length of this beautiful substring by summing the lengths of its sequences (`a[1] + b[1] + c[1] + d[1] + e[1]`) and updates `ans` if this length is greater than the current `ans`.
6. **Return the result**: After iterating through all possible substrings, the algorithm returns the maximum length found (`ans`), which represents the length of the longest beautiful substring.

Here are the key algorithms, data structures, and patterns used:

- **Data Structure (List of Tuples)**: The list `arr` of tuples is crucial for keeping track of sequences of the same character and their lengths. This allows for efficient access and analysis of contiguous segments that may form parts of a beautiful substring.
- **Two-Pointer Technique**: The algorithm uses two pointers (`i` and `j`) to identify the sequences of identical characters. The first pointer `i` marks the start of a sequence, while the second pointer `j` moves ahead to find the end of that sequence.
- **Sliding Window**: By checking slices of 5 consecutive elements in `arr`, the algorithm effectively uses a sliding window of size 5 to identify potential beautiful substrings.
- **Greedy Approach**: By always updating `ans` with the maximum length found, we ensure that by the end of the algorithm, we have greedily found the longest beautiful substring.

Taken together, this approach efficiently identifies the longest beautiful substring by combining sequence aggregation and step-by-step analysis with a logical check for the "beautiful" conditions.

Example Walkthrough

Let's use a small example string `"aeeiouaeiou"` to illustrate the solution approach described above. We'll walk through each step of the solution using this string.

1. **Initialize a list to store character sequence information**: We start with an empty list `arr`.
2. **Loop through the string to fill the sequence list**: We would start at the first character `a` and notice that the next character is `e`, so we have a sequence of `"a"` with a length of `1`. We add `('a', 1)` to our `arr` list.

   Continuing this process, we would get the following sequences:

   - Sequence of `"e"` with length `2`, so we add `('e', 2)` to our `arr`.
   - Sequence of `"i"` with length `1`, so we add `('i', 1)` to our `arr`.
   - Sequence of `"o"` with length `1`, since `o` and `u` are different, we add `('o', 2)` for the two `o`s and then `('u', 1)` for the single `u`.
   - This process repeats until the end of the string, resulting in our `arr` being `[('a', 1), ('e', 2), ('i', 1), ('o', 2), ('u', 1), ('a', 1), ('e', 1), ('i', 1), ('o', 1), ('u', 1)]`.
3. **Initialize a variable to keep track of the answer**: We set `ans = 0` as we have not found any beautiful substrings yet.
4. **Search for beautiful substrings**: We start iterating over `arr` to find sequences of five consecutive character sequences that match `'aeiou'`.
5. **Update the maximum length if a beautiful substring is found**: As we iterate, we check slices of `arr` such as `arr[0:5]` which would give us `[('a', 1), ('e', 2), ('i', 1), ('o', 2), ('u', 1)]`. This is a beautiful sequence because the characters are in the correct `'aeiou'` order. We, therefore, calculate the length of this beautiful substring as `1 + 2 + 1 + 2 + 1 = 7` and update `ans` to `7` because it is greater than the current `ans`.

   When we reach the slice `arr[5:10]`, which is `[('a', 1), ('e', 1), ('i', 1), ('o', 1), ('u', 1)]`, we do not have a beautiful sequence since the characters are not in the correct order.
6. **Return the result**: After iterating through the entire list `arr`, the algorithm finds that the longest beautiful substring length is `7`, and that is what it returns.

Therefore, for the example string `"aeeiouaeiou"`, the algorithm would correctly identify the longest beautiful substring `"aeeiou"` and return its length, `7`.

Python Solution

```python
class Solution:
    def longestBeautifulSubstring(self, word: str) -> int:
        # Initialize a list to store tuples of characters and their consecutive counts
        consecutive_chars = []
        # Start at the first character
        index = 0

        # Iterate through the word to group consecutive characters together
        while index < len(word):
            # Count of the current character
            start_index = index
            # Move index forward while the characters are the same
            while index < len(word) and word[index] == word[start_index]:
                index += 1
            # Append the character and its consecutive count to the list
            consecutive_chars.append((word[start_index], index - start_index))

        # Set initial answer to 0
        max_length = 0

        # Iterate through the grouped character list to find beautiful substrings
        for i in range(len(consecutive_chars) - 4):  # We need at least 5 different vowels
            # Unpack the next five elements in the list
            first_vowel, second_vowel, third_vowel, fourth_vowel, fifth_vowel = consecutive_chars[i : i + 5]
            # Check if current sequence forms "aeiou"
            if (first_vowel[0] == 'a' and second_vowel[0] == 'e' and third_vowel[0] == 'i' and
                fourth_vowel[0] == 'o' and fifth_vowel[0] == 'u'):
                # Calculate the total length of the current beautiful substring
                current_length = (first_vowel[1] + second_vowel[1] + third_vowel[1] + fourth_vowel[1] + fifth_vowel[1])
                # Update max_length if the current beautiful substring
                max_length = max(max_length, current_length)

        # Return the length of the longest beautiful substring found
        return max_length
```

Java Solution

```java
class Solution {

    // Method to find the length of the longest beautiful substring in the input string
    public int longestBeautifulSubstring(String word) {
        int wordLength = word.length(); // Store the length of the word
        List<CharGroup> charGroups = new ArrayList<>(); // List to store groups of consecutive identical characters

        // Loop through the string and group consecutive identical characters
        for (int i = 0; i < wordLength;) {
            int j = i + 1;
            // Find the end index of the group of identical characters
            while (j < wordLength && word.charAt(j) == word.charAt(i)) {
                ++j;
            }
            // Add the group to the list
            charGroups.add(new CharGroup(word.charAt(i), j - i));
            i = j; // Move to the next group
        }

        int maxBeautyLength = 0; // Variable to track the maximum length of a beautiful substring

        // Iterate through the list of char groups to find the longest beautiful substring
        for (int i = 0; i < charGroups.size() - 4; ++i) {
            // Get five consecutive char groups
            CharGroup a = charGroups.get(i),
                      b = charGroups.get(i + 1),
                      c = charGroups.get(i + 2),
                      d = charGroups.get(i + 3),
                      e = charGroups.get(i + 4);
            // Check if the groups form a sequence 'a', 'e', 'i', 'o', 'u'
            if (a.character == 'a' && b.character == 'e' && c.character == 'i' &&
                d.character == 'o' && e.character == 'u') {
                // Calculate the total length of the beautiful substring and update the max length
                maxBeautyLength = Math.max(maxBeautyLength, a.count + b.count + c.count + d.count + e.count);
            }
        }

        return maxBeautyLength; // Return the maximum length found
    }
}

// Helper class to represent a group of consecutive identical characters
class CharGroup {
    char character; // The character in the group
    int count; // The count of how many times the character is repeated

    // Constructor for the helper class
    CharGroup(char character, int count) {
        this.character = character;
        this.count = count;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int longestBeautifulSubstring(string word) {
        // Vector to store pairs of characters and their consecutive frequencies.
        vector<pair<char, int>> charFrequencies;
        int length = word.size();

        // Convert the word into pairs of characters and their consecutive counts.
        for (int i = 0; i < length;) {
            int j = i + 1;
            while (j < length && word[j] == word[i]) {
                ++j;
            }
            charFrequencies.push_back({word[i], j - i});
            i = j; // Move to the next unique character.
        }

        int maxBeautyLength = 0; // To store the length of the longest beautiful substring

        // Loop through the charFrequencies array to find all possible beautiful substrings.
        for (int i = 0; i < charFrequencies.size() - 4; ++i) {
            auto [currentChar, currentFreq] = charFrequencies[i];
            auto [nextChar1, freq1] = charFrequencies[i + 1];
            auto [nextChar2, freq2] = charFrequencies[i + 2];
            auto [nextChar3, freq3] = charFrequencies[i + 3];
            auto [nextChar4, freq4] = charFrequencies[i + 4];

            // Check if we have a sequence 'aeiou', denoting a beautiful substring.
            if (currentChar == 'a' && nextChar1 == 'e' && nextChar2 == 'i' &&
                nextChar3 == 'o' && nextChar4 == 'u') {
                // Calculate the length of the beautiful substring and update maxBeautyLength.
                int beautyLength = currentFreq + freq1 + freq2 + freq3 + freq4;
                maxBeautyLength = max(maxBeautyLength, beautyLength);
            }
        }

        // Return the length of the longest beautiful substring found.
        return maxBeautyLength;
    }
};
```

Typescript Solution

```typescript
function longestBeautifulSubstring(word: string): number {
    // Array to store pairs of characters and their consecutive frequencies.
    const charFrequencies: [character: string, frequency: number][] = [];
    const length: number = word.length;

    // Convert the word into objects with characters and their consecutive counts.
    for (let i = 0; i < length; ) {
        let j = i;
        while (j < length && word[j] === word[i]) {
            ++j;
        }
        charFrequencies.push([word[i], frequency: j - i ]);
        i = j; // Move to the next unique character.
    }

    let maxBeautyLength: number = 0; // To store the length of the longest beautiful substring

    // Iterate through the charFrequencies array to find all possible beautiful substrings.
    for (let i = 0; i < charFrequencies.length - 4; ++i) {
        const [currentChar, currentFreq] = charFrequencies[i];
        const [nextChar1, freq1] = charFrequencies[i + 1];
        const [nextChar2, freq2] = charFrequencies[i + 2];
        const [nextChar3, freq3] = charFrequencies[i + 3];
        const [nextChar4, freq4] = charFrequencies[i + 4];

        // Check if we have a sequence 'aeiou', denoting a beautiful substring.
        if (currentChar === 'a' && nextChar1 === 'e' &&
            nextChar2 === 'i' && nextChar3 === 'o' &&
            nextChar4 === 'u') {
            // Calculate the length of the beautiful substring and update maxBeautyLength.
            const beautyLength = currentFreq + freq1 + freq2 + freq3 + freq4;
            maxBeautyLength = Math.max(maxBeautyLength, beautyLength);
        }
    }

    // Return the length of the longest beautiful substring found.
    return maxBeautyLength;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be analyzed in the following steps:

1. Constructing the `arr` list: This involves a single pass through the input string `word` with a pair of pointers `i` and `j`. For each unique character in the word, the loop checks for consecutive occurrences and adds a tuple (`character, count`) to `arr`. This operation has a time complexity of $O(n)$ where `n` is the length of the input string since each character is considered exactly once.
2. Looping through `arr` for finding the longest beautiful substring: The second loop runs with an upper limit of `len(arr) - 4`, and for each iteration, it checks a fixed sequence of 5 elements (not considering nested loops). The check and `max` calls are $O(1)$ operations. The number of iterations depends on the number of unique characters in `word`, but since it's strictly less than `n`, the loop has a time complexity of $O(n)$.

Combining both parts, the overall time complexity is $O(n) + O(n) = O(n)$.

Space Complexity

The space complexity is determined by additional space used apart from the input:

1. The `arr` list: In the worst case, if every character in `word` is unique, `arr` would have `n` tuples. Therefore, the space complexity due to `arr` is $O(n)$.
2. Constant space for variables `i`, `j`, and `ans`, which doesn't depend on the size of the input.

Hence, the overall space complexity of the code is $O(n)$.