830. Positions of Large Groups

Easy String

Leetcode Link

Problem Description

The given problem revolves around identifying the 'large' groups of consecutive identical characters in a string `s`. A 'large' group is defined as a sequence of the same character that appears at least three times consecutively. The goal is to find the starting index and the ending index for each of these large groups.

Consider this example: In the string `s = "abbxxxxzyy"`, the following groups are formed: `"a"`, `"bb"`, `"xxxx"`, `"z"`, and `"yy"`. Out of these, only `"xxxx"` qualifies as a large group because it contains the same character ('x') four times in a row. The interval for this group is `[3, 6]`, where `3` is the starting index, and `6` is the ending index of the group within the string.

The expected output is a list of intervals with each interval representing a large group and the list is sorted by the starting index of each interval.

Intuition

To solve this problem, the approach is to iterate over the string while keeping track of the start of a potential large group. Whenever we encounter a different character or reach the end of the string, we check if the current character sequence qualifies as a large group by ensuring the sequence length is at least three characters long. If it does, we record the start and the end indices of this group. Next, we update our tracker to the current character's index and continue scanning until we have examined all characters in the string.

Iterating only once through the string results in an efficient solution with a linear time complexity (O(n)), where n is the length of the string. We can assert that the solution is efficient because we're only scanning the string once without any nested loops, making it suitable for large input strings as well.

The key takeaway is that we maintain the current character sequence length and start index while traversing the string and only record intervals that satisfy the 'large group' criteria.

Solution Approach

The solution's implementation utilizes a two-pointer pattern, which is often used in problems involving arrays or strings where we need to track subarrays or substrings based on a certain condition. Here, the two pointers are denoted as `i` (the start pointer) and `j` (the end pointer).

The approach begins with initializing the starting pointer `i` to 0 and an empty list `ans` to collect the intervals representing the large groups.

The main algorithm can be described as follows:

1. Begin iterating over the string with the starting pointer `i`, aiming to pinpoint the start of a character sequence.
2. Create an inner loop that increments the end pointer `j` for as long as the character at position `j` is the same as the character at the start pointer `i`.
3. Upon exit from the inner loop, check if the difference between `j` and `i` is greater than or equal to 3, indicating a large group.
4. If it is indeed a large group, append the interval `[i, j - 1]` to the list `ans`, since `j - 1` is the end index of the large group.
5. Finally, update the start pointer `i` to `j` to commence scanning for the next potential group.

The code maintains a while loop that will run until `i` reaches the end of the string (`i < n`). Each time a large group is found, the interval is appended to the list before resetting `i` to continue searching from the end of the last found group.

The use of the continuous while loop paired with the two-pointer technique ensures all large groups are identified without repetitively checking characters, which efficaciously reduces the time complexity.

Once the while loop is completed, the function returns `ans`, which now contains all intervals of the identified large groups, arranged by their start indices as per the requirement.

Here is an excerpt showcasing the two-pointer technique in the given code:

```
while i < n:           # Main loop iterating through the string
    j = i              # Initializing end pointer
    while j < n and s[j] == s[i]:  # Inner loop to increment 'j' as long as characters match
        j += 1
    if j - i >= 3:     # If the length of the group is at least 3, it is a large group
        ans.append([i, j - 1])  # Append interval to answer list as a large group is found
    i = j              # Move start pointer 'i' to the end of the last found group
```

This solution is efficient as it involves a single pass of the input string, making the time complexity O(n), where n is the size of the string.

Example Walkthrough

Let's walk through an example to illustrate the solution approach:

Consider the string `s = "aabbbccdeee"`. We need to find the large groups of consecutive identical characters, where a large group is defined as a sequence with at least three occurrences of the same character.

1. Initialize the start pointer `i` to 0 and create an empty list `ans` for storing our intervals.
2. Initiate the main while loop, where `i` starts at index 0.
3. The end pointer `j` is also initialized to the same value as `i`.
4. Start the inner while loop that compares characters at positions `j` and `i` and increments `j` as long as they are identical. For the first loop, we compare characters `'a'` at index 0 and 1, and since they are not identical, `j` stops at 1.
5. No interval is added to `ans` after the first loop since `j - i < 3`.
6. Now, `i` is set to `j`, and our pointers are `i = 1`, `j = 1`. The comparison starts with `'a'` and `'b'`. Since they are different characters, we immediately look for the next sequence.
7. Next iteration, `i` is still at 1, and `'b'` at position 1 matches `'b'` at positions 2 and 3. The inner loop increments `j` to 4.
8. Now `j - i >= 3`—it's a large group, so we add the interval `[1, 3]` to `ans`.
9. Update `i` to the current value of `j`, which is 4.
10. Continue this process for the rest of the string.
    - `i = 4`, `j = 4`: Match `'c'` with subsequent `'c'` until `j = 6`. `j - i = 2`, not a large group.
    - `i = 6`, `j = 6`: Skip `'d'` as it's a single character.
    - `i = 7`, `j = 7`: Characters `'e'` at index 7, 8, and 9 match. `j` moves to 10. `j - i >= 3`, we add the interval `[7, 9]` to `ans`.
11. Finally, our list `ans` contains `[[1, 3], [7, 9]]`, representing the large groups `'bbb'` and `'eee'` with their respective starting and ending indices.

The implementation efficiently identifies the large groups in a single traversal of the string `s`. The output list is sorted by starting index because the string is traversed from left to right, naturally maintaining the order.

Python Solution

```python
class Solution:
    def largeGroupPositions(self, s: str) -> List[List[int]]:
        # Initialize start index `start` and length of the string `length`
        start, length = 0, len(s)
        # Initialize an empty list to store the answer
        answer = []

        # Iterate over the string characters by index
        while start < length:
            # Initialize the end index `end` at the same position as start
            end = start
            # Move `end` forward as long as the subsequent characters are the same as s[start]
            while end < length and s[end] == s[start]:
                end += 1
            # If the length of the group is 3 or more, add it to the answer list
            if end - start >= 3:
                answer.append([start, end - 1])
            # Update start to the next character group
            start = end

        # Return the final list of positions of large groups
        return answer

```

Java Solution

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

class Solution {

    // Method to find the starting and ending indices of all large groups.
    public List<List<Integer>> largeGroupPositions(String s) {
        int strLength = s.length(); // Length of the input string.
        int startIndex = 0; // Initialize the starting index of a group.
        List<List<Integer>> largeGroups = new ArrayList<>(); // Initialize the list to store the result.

        // Iterate over the string to find large groups.
        while (startIndex < strLength) {
            int endIndex = startIndex; // Initialize the end index of the current group.
            // Increase the endIndex as long as the current character is the same as the start character.
            while (endIndex < strLength && s.charAt(endIndex) == s.charAt(startIndex)) {
                ++endIndex;
            }
            // Check if the current group is a large group (i.e., has a length of 3 or more).
            if (endIndex - startIndex >= 3) {
                // Add the start and end indices (end index is exclusive so subtract 1) of the large group to the result.
                largeGroups.add(Arrays.asList(startIndex, endIndex - 1));
            }
            startIndex = endIndex; // Move the startIndex to the end of the current group to start checking the next group.
        }
        return largeGroups; // Return the list of all large groups found.
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>

class Solution {
public:
    std::vector<std::vector<int>> largeGroupPositions(std::string s) {
        int length = s.size();          // Get the size of the input string
        int startIndex = 0;             // Initialize the starting index of a group
        std::vector<std::vector<int>> largeGroups; // Result vector for storing large group positions

        // Iterate over the string to identify groups
        while (startIndex < length) {
            int endIndex = startIndex;  // Initialize the end index of the group to the current start index

            // Advance the end index as long as characters match the character at the start index
            while (endIndex < length && s[endIndex] == s[startIndex]) {
                ++endIndex;
            }

            // Check if the identified group is a large group (3 or more characters)
            if (endIndex - startIndex >= 3) {
                // Add the start and end indices (exclusive of the end) of the large group to the result
                largeGroups.push_back({startIndex, endIndex - 1});
            }

            startIndex = endIndex; // Move the start index to the end of the current group for the next iteration
        }

        return largeGroups; // Return the vector of large groups
    }
};

```

Typescript Solution

```typescript
function largeGroupPositions(s: string): number[][] {
    let length: number = s.length; // Get the length of the input string
    let startIndex: number = 0; // Initialize the starting index of a group
    let largeGroups: number[][] = []; // Result array for storing large group positions

    // Iterate over the string to identify groups
    while (startIndex < length) {
        let endIndex: number = startIndex; // Initialize the end index of the group to the current start index

        // Advance the end index as long as characters match the character at the start index
        while (endIndex < length && s.charAt(endIndex) === s.charAt(startIndex)) {
            endIndex++;
        }

        // Check if the identified group is a large group (3 or more characters)
        if (endIndex - startIndex >= 3) {
            // Add the start and end indices (end is exclusive) of the large group to the result
            largeGroups.push([startIndex, endIndex - 1]);
        }

        startIndex = endIndex; // Move the start index to the end of the current group for the next iteration
    }

    return largeGroups; // Return the array of large groups
}

```

Time and Space Complexity

Time Complexity

The given Python function scans through the string `s` once. The inner while-loop advances the index `j` as long as consecutive characters are equal to `s[i]`, and the outer while-loop is responsible for iterating over each character, but thanks to the inner while-loop skipping groups of the same character, every character is visited at most once.

The time complexity for scanning the string is `O(n)`, where `n` is the length of the string, since in the worst case, we have to look at each character.

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

Space Complexity

The space complexity is determined by the space needed to store the output, which is the list `ans`.

In the worst case, if we have a string where every three characters form a group (for example, `"aaabbbccc"`), the number of groups (and thus the number of sublists in `ans`) will be approximately `n/3`. Each group is represented by a pair of indices (which takes constant space), so the space required grows linearly with the number of groups.

Thus, the space complexity is `O(n)` where `n` is the length of the string, as the size of the `ans` list is proportional to the number of large groups in the string.