1221. Split a String in Balanced Strings

Easy Greedy String Counting

Leetcode Link

Problem Description

A balanced string is a string that contains an equal number of 'L' and 'R' characters. The problem asks to split a given balanced string `s` into the maximum number of balanced substrings. A substring is considered balanced if it contains an equal number of 'L' and 'R' characters. The challenge is to find the maximum number of balanced substrings that can be obtained from the input string.

Intuition

To solve this problem, we need to iterate through the string and keep track of the number of 'L' and 'R' characters we've encountered. The idea is to increase a counter every time we encounter 'L' and decrease it every time we encounter 'R'. When this counter returns to zero, it means we have found an equal number of 'L' and 'R' characters, hence a balanced substring.

By initializing a variable `l` to zero, we use it as a counter to keep track of the balance between 'L' and 'R' characters. If we encounter 'L', we increment `l`, and if we encounter 'R', we decrement `l`. Each time `l` becomes zero, it's an indication that the substring from the start to the current position is balanced, and we increment our answer `ans` by one.

This approach works because we are given that the original string `s` is balanced. Therefore, as we process the string from left to right, any time `l` is zero, we have found a balanced substring, and we can safely split the string at that point and start counting a new potential balanced substring.

The algorithm ends when we've gone through the entire string, and the final value of `ans` is the maximum number of balanced substrings we can obtain.

Solution Approach

The implementation of the solution uses a simple but effective algorithm that requires only a single pass through the string. Its simplicity is derived from its reliance on a single integer counter and the characteristics of the string being balanced. Let's walk through the code to understand this better.

In Python, we define a class `Solution` with a method `balancedStringSplit` that takes a single argument, the string `s`. The method is structured as follows:

1. We initialize two integers `ans` and `l` to `0`. `ans` will hold the final count of the balanced substrings, and `l` will be used as a counter to track the balance between 'L' and 'R' characters within a substring.
2. We iterate over each character `c` in the string `s` with a `for` loop.
3. Inside the loop, we check if `c` is 'L'. If so, we increment `l`, otherwise (meaning `c` is 'R'), we decrement `l`.
4. After updating `l` for the current character, we check if `l` has become `0`. If it has, this indicates that we've encountered an equal number of 'L' and 'R' characters up to the current point in the string, forming a balanced substring. We increment `ans` to count this new balanced substring.
5. Once the loop has finished, we have iterated over the entire string, and `ans` contains the maximum number of balanced substrings we could form.

Here is the Python code that implements this algorithm:

```python
class Solution:
    def balancedStringSplit(self, s: str) -> int:
        ans = l = 0
        for c in s:
            if c == 'L':
                l += 1
            else:
                l -= 1
            if l == 0:
                ans += 1
        return ans
```

There are no additional data structures needed for this solution, and it works in $O(n)$ time, where n is the length of the string, because we are making just one pass through it. The space complexity is $O(1)$ as we are only using a fixed amount of additional space (the two integers `ans` and `l`).

This algorithm is based on the pattern that a substring is balanced if and only if the number of 'L's is equal to the number of 'R's. Given that the entire string is balanced, we know that each 'L' will eventually be matched by an 'R'. Therefore, the key insight is to count the number of balanced substrings incrementally as we traverse the string.

Example Walkthrough

Let's consider a simple example to illustrate the solution approach. Suppose we are given the balanced string `s = "RLLRRLRL"`. We want to determine the maximum number of balanced substrings we can obtain from this string.

1. We initialize `ans = 0` (the count of balanced substrings) and `l = 0` (the balance counter). The string `s` is `"RLLRRLRL"`.
2. As we iterate through the characters of `s`, we apply the following logic:
   - We encounter 'R': `l` is decremented by 1 (`l = -1`).
   - Next is 'L': `l` is incremented by 1 (`l = 0`).
   - Since `l` is now 0, we have found our first balanced substring "RL". `ans` is incremented by 1 (`ans = 1`).
3. Continuing the iteration:
   - We encounter 'L': `l` is incremented by 1 (`l = 1`).
   - Next is 'L': `l` is incremented by 1 again (`l = 2`).
   - Then 'R': `l` is decremented by 1 (`l = 1`).
   - And 'R': `l` is decremented by 1 again (`l = 0`).
   - As `l` has returned to 0, we increment `ans` by 1 (`ans = 2`).

   At this stage, we have identified another balanced substring: "LLRR".
4. For the remaining characters:
   - We have 'L': increment `l` by 1 (`l = 1`).
   - Then 'R': decrement `l` by 1 (`l = 0`).
   - Increment `ans` (now `ans = 3`) since we've found another balanced substring "LR".

   Continuation of the sequence:
   - Encounter 'L': `l` is incremented by 1 (`l = 1`).
   - And finally 'R': `l` is decremented back to 0 (`l = 0`).
   - Increment `ans` once more (final `ans = 4`) as the last pair "LR" forms a balanced substring.
5. Now that we've processed the entire string `s`, `ans` holds the value 4, which represents the maximum number of balanced substrings.

The substring splits we've found are "RL", "LLRR", "LR", and "LR". Each of these substrings has an equal number of 'L' and 'R' characters, thus meeting the criteria for balanced substrings. The method `balancedStringSplit` would return 4 for the input `"RLLRRLRL"`, and this concludes our example walkthrough using the solution approach.

Python Solution

```python
class Solution:
    def balancedStringSplit(self, s: str) -> int:
        # Initialize count of balanced substrings and a balance counter
        balanced_count = 0
        balance = 0

        # Iterate over each character in the string
        for char in s:
            # If the character is 'L', increment the balance counter
            if char == 'L':
                balance += 1
            # If the character is 'R', decrement the balance counter
            else:
                balance -= 1

            # Check if the substring is balanced
            if balance == 0:
                # Increment the count of balanced substrings
                balanced_count += 1

        # Return the total count of balanced substrings
        return balanced_count
```

Java Solution

```java
class Solution {
    // Method to count the number of balanced strings in the input string 's'.
    // A balanced string has an equal number of 'L' and 'R' characters.
    public int balancedStringSplit(String s) {
        int balanceCount = 0; // To store the number of balanced strings found
        int balance = 0; // A variable to track the balance between 'L' and 'R' characters

        // Loop through each character in the string
        for (char c : s.toCharArray()) {
            // Increment balance when 'L' is found
            if (c == 'L') {
                balance++;
            // Decrement balance when 'R' is found
            } else if (c == 'R') {
                balance--;
            }

            // When balance is zero, a balanced string is found
            if (balance == 0) {
                balanceCount++;
            }
        }

        // Return the total number of balanced strings
        return balanceCount;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // This function counts how many times the input string can be split
    // into balanced strings, where "balanced" means the number of 'L's and 'R's
    // in the substring are equal.
    int balancedStringSplit(string s) {
        int countBalanced = 0; // To keep track of the count of balanced strings
        int balanceFactor = 0; // To keep track of the balance between 'L' and 'R'

        // Iterate over each character in the string
        for (char c : s) {
            // Increment balance factor for 'L' and decrement for 'R'
            if (c == 'L') {
                balanceFactor++;
            } else { // c == 'R'
                balanceFactor--;
            }

            // When the balance factor is zero, we have a balanced string
            if (balanceFactor == 0) {
                countBalanced++; // Increment the number of balanced strings
            }
        }

        // Return the total number of balanced strings
        return countBalanced;
    }
};
```

Typescript Solution

```typescript
/**
 * Function to count the number of balanced strings that can be split.
 * Balanced strings are those where the quantity of 'L' and 'R' characters is the same.
 *
 * @param {string} inputString - The string to be checked for balanced splits.
 * @return {number} - The count of balanced strings that can be split.
 */
const balancedStringSplit = (inputString: string): number => {
    let balancedCount = 0; // Counter for balanced splits
    let balance = 0; // Helper to keep track of the current balance between 'L' and 'R'

    // Iterate through the string to check for balanced segments.
    for (let character of inputString) {
        balance += (character == 'L') ? 1 : -1;
        // If balance is 0, a balanced segment is found.
        if (balance === 0) {
            balancedCount++;
        }
    }

    return balancedCount; // Return the total number of balanced strings.
};

// Example usage:
// let result = balancedStringSplit("RLRRLLRLRL");
// console.log(result); // Outputs: 4, because there are four balanced substrings "RL", "RRLL", "RL", "RL"
```

Time and Space Complexity

Time Complexity

The time complexity of the code is $O(n)$, where n is the length of the input string `s`. This is because the code iterates through each character in the string exactly once, performing a constant amount of work for each character with simple arithmetic operations and a comparison.

Space Complexity

The space complexity of the code is $O(1)$. The code uses a fixed number of integer variables (`ans` and `l`) regardless of the input size. These variables take up a constant amount of space and do not scale with the size of the input string.