1422. Maximum Score After Splitting a String

Easy String

Problem Description

In this problem, we are given a string `s` consisting of characters '0' and '1'. Our goal is to split this string into two non-empty parts, a left and a right substring, such that the score obtained from the split is maximized. The score is calculated as the number of '0' characters in the left substring added to the number of '1' characters in the right substring. We need to determine the maximum score we can achieve by splitting the string `s`. It's important to note that the split must result in non-empty substrings, which means we can split the string anywhere between the first and last character.

Intuition

To solve this problem, we need to understand that the split will always be between two characters in the string. In terms of index `i`, the left substring will include characters from index `0` to `i`, and the right substring will include characters from index `i+1` to the end. Our goal is to maximize the sum of zeros in the left substring and ones in the right substring.

A straightforward approach would be to check all possible splits and calculate the score for each, but that would be inefficient. Instead, we can take advantage of the fact that any split only moves one character from one substring to the other, thus changing the score by either 1 or -1. This allows us to iteratively calculate the score for all possible splits in a single pass through the string after an initial count.

Here's the step-by-step intuition:

1. Start by calculating the initial score, which is the number of '0's in the first position plus the number of '1's in the rest of the string (since the first character must be in the left substring for it to be non-empty).
2. Initialize the maximum score `ans` with this initial score.
3. Iterate over the string starting from the second character to the second-to-last character, updating the current score `t` and maximum score `ans` as follows:
   - If we find a '0', it means moving the split to the right would bring one more '0' to the left substring, so we increase `t` by 1.
   - If we find a '1', it means moving the split to the right would take one '1' out of the right substring, so we decrease `t` by 1.
   - After each update, compare `t` with the current `ans` and update `ans` if `t` is higher.
4. Once we've considered all splits, the `ans` variable will hold the maximum score possible.

Solution Approach

The solution utilizes a simple yet efficient algorithm that takes advantage of the cumulative property of scores during the split process. Let's delve into the key aspects of the implementation:

1. **Initializing Scores**:
   - The initial score `t` is computed by checking if the first character is '0' and adding it to the count of '1's in the rest of the string. This is because we want all '0's to be in the left substring and '1's to be in the right one to maximize the score.
   - The Python expression `(s[0] == '0')` returns `True` if the first character is '0' and `False` otherwise. Since `True` equates to `1` and `False` to `0` in Python, this expression conveniently adds `1` to the count if the first character is '0'.
   - The count of '1's in the right substring is calculated using `s[1:].count('1')`, where `s[1:]` is the substring excluding the first character.
2. **Iterating over Characters**:
   - A `for` loop is used to iterate through each character in the string starting from index 1 and ending at the second-last character, since we are only interested in valid splits (non-empty left and right substrings).
   - During each iteration, the variable `t` is updated to reflect the new score if the split were to occur right after the current character. If `s[i]` is '0', `t` is incremented by 1 because we'd have one more '0' in the left substring. Conversely, if `s[i]` is '1', `t` is decremented by 1 because we'd have one less '1' in the right substring.
3. **Updating Maximum Score**:
   - The maximum score so far is kept in the variable `ans`. After each score update, `ans` is set to the maximum of itself and the current score `t` by executing `ans = max(ans, t)`.
4. **Return Final Score**:
   - After the loop completes, the maximum score `ans` is returned.

By using a single pass through the string and updating scores incrementally, this solution achieves a time complexity of O(n), where n is the length of the string. The space complexity is O(1) since only a fixed number of variables are used.

We use no additional data structures; instead, we capitalize on the fact that iterating over the string while tracking the current score and the maximum score suffices to solve the problem. The pattern used here involves greedy local optimizations at each character index that leads to a global optimization, i.e., the maximum score.

Example Walkthrough

Let's use a small example string `s = "011101"` to illustrate the solution approach step-by-step:

1. **Initializing Scores**:
   - We start by calculating the initial score `t` for the first possible split, which will have all characters in the left substring except the first one on the right side.
   - With our example, the first character is '0', so according to the logic, `(s[0] == '0')` will be `True`, and thus `t` starts at 1.
   - Then, we count the number of '1's in the rest of the string (`"11101"`), which is 4. So, the initial score `t` is `1 + 4 = 5`.
   - We also initialize the maximum score `ans` to this initial value, `ans = 5`.
2. **Iterating over Characters**:
   - We start processing the string from the second character to the second-to-last character.
   - As we move to the next character, which is '1', the score `t` does not change as per the rule because '1' is already in the right substring. Therefore, `t` remains 5 and `ans` remains 5.
   - The next character is '1' again, and nothing changes for the same reason, so `t` and `ans` are still 5.
   - The fourth character is '1', and again, nothing changes, so `t` and `ans` remain 5.
   - Now, the fifth character is '0'. If we were to split here, the '0' would move to the left substring. This means we add 1 to `t`, making `t = 5 + 1 = 6`. We compare this with `ans` and since `t` is greater, we update `ans` to 6.
   - Moving to the final character, '1', if we were to split here, we would subtract 1 from `t` because the '1' would move out of the right substring. This gives us `t = 6 - 1 = 5`. `ans` remains 6 since it was not exceeded.
3. **Updating Maximum Score**:
   - Throughout the process, `ans` is updated to the maximum score encountered during each iteration. The maximum score that we calculated in the example was 6.
4. **Return Final Score**:
   - After the loop, we conclude that the maximum score that can be achieved by splitting the string `s = "011101"` is 6. This is the value stored in `ans`, which is what our function would return.

Solution Implementation

Python

```python
class Solution:
    def maxScore(self, s: str) -> int:
        # Initial score considers the first character '0' as +1 and counts '1's in the rest of the string.
        score = (s[0] == '0') + s[1:].count('1')
        # Initialized answer with the initial score.
        max_score = score

        # Loop through the string starting from the second character to the second to last.
        for i in range(1, len(s) - 1):
            # If the character is '0', increment score, if '1', decrement it.
            score += 1 if s[i] == '0' else -1
            # Update max score if the current score is greater.
            max_score = max(max_score, score)

        # Return the maximum score achieved.
        return max_score
```

Java

```java
class Solution {
    public int maxScore(String s) {
        // Initialize total score for the first partition
        int totalScore = 0;
        // If the first character is '0', increase the score by 1
        if (s.charAt(0) == '0') {
            totalScore++;
        }
        // Calculate initial score for '1's in the string, skipping the first character
        for (int i = 1; i < s.length(); ++i) {
            if (s.charAt(i) == '1') {
                totalScore++;
            }
        }
        // The best score is initially set to the score from the initial partition
        int maxScore = totalScore;

        // Iterate through the string to find partitions and track the highest score
        for (int i = 1; i < s.length() - 1; ++i) {
            // If the current character is '0', increase the total score
            // If it is '1', decrease the total score
            totalScore += s.charAt(i) == '0' ? 1 : -1;
            // Update the maximum score if the current total score is greater
            maxScore = Math.max(maxScore, totalScore);
        }
        // Return the highest score found amongst all possible partitions
        return maxScore;
    }
}
```

C++

```cpp
class Solution {
public:
    int maxScore(string s) {
        // Initialize totalScore to 0. It will represent the total score while traversing the string.
        int totalScore = 0;

        // If the first character is '0', we increment totalScore as we get one point for it.
        if (s[0] == '0') {
            ++totalScore;
        }

        // Add to totalScore the number of '1' in the string, except for the first character.
        for (int i = 1; i < s.size(); ++i) {
            totalScore += s[i] == '1';
        }

        // Initialize maxScore with the totalScore calculated so far.
        int maxScore = totalScore;

        // Start traversing the string from the second character to the second-to-last.
        for (int i = 1; i < s.size() - 1; ++i) {
            // Increment totalScore if it's a '0', else decrement it for a '1'.
            totalScore += s[i] == '0' ? 1 : -1;

            // Update maxScore with the higher value between maxScore and totalScore.
            maxScore = max(maxScore, totalScore);
        }

        // Return the maximum score found.
        return maxScore;
    }
};
```

TypeScript

```typescript
function maxScore(s: string): number {
    const stringLength = s.length; // Store the length of the input string
    let maxResult = 0;             // Initialize the variable to hold the maximum score
    let currentScore = 0;          // Initialize the variable to hold the current score

    // First, we check if the first character is '0' to possibly increase the current score,
    // because we are looking for the maximum number of '0's in the left partition
    // and the maximum number of '1's in the right partition of the string after a split.
    if (s[0] === '0') {
        currentScore++;
    }

    // Calculate the score of 1's in the right partition by iterating over the string
    // starting at index 1 (second character) because the left partition will have at least
    // one character.
    for (let i = 1; i < stringLength; i++) {
        if (s[i] === '1') {
            currentScore--;
        }
    }

    // Initially set the maximum score to the score calculated above
    maxResult = Math.max(maxResult, currentScore);

    // Loop through s, starting from the second character to the second-to-last character
    // since we need to split s into two non-empty partitions.
    for (let i = 1; i < stringLength - 1; i++) {
        // If the current character is '0', we add one to the current score
        // because we are effectively moving a '0' from the right partition (initially the whole string sans the first character)
        // to the left partition, increasing the left partition's score.
        if (s[i] === '0') {
            currentScore++;
        // If the current character is '1', we subtract one from the current score
        // because we are moving a '1' from the right partition to the left one,
        // decreasing the right partition's score.
        } else if (s[i] === '1') {
            currentScore--;
        }
        // Update the maximum result if the current score is greater
        maxResult = Math.max(maxResult, currentScore);
    }

    // Finally return the maximum score found, which corresponds to the
    // best place to split string s into two partitions
    return maxResult;
}
```

```python
class Solution:
    def maxScore(self, s: str) -> int:
        # Initial score considers the first character '0' as +1 and counts '1's in the rest of the string.
        score = (s[0] == '0') + s[1:].count('1')
        # Initialized answer with the initial score.
        max_score = score

        # Loop through the string starting from the second character to the second to last.
        for i in range(1, len(s) - 1):
            # If the character is '0', increment score, if '1', decrement it.
            score += 1 if s[i] == '0' else -1
            # Update max score if the current score is greater.
            max_score = max(max_score, score)

        # Return the maximum score achieved.
        return max_score
```

Time and Space Complexity

Time Complexity

The time complexity is determined by how many operations the algorithm performs in relation to the input size, which is the length of the string `s` in this case.

- First, the algorithm initializes `ans` and `t` with the count of '1's in the substring `s[1:]` and checks if `s[0]` is '0'. This takes `O(n)` time where `n` is the length of `s`, as `count('1')` iterates through the string once.
- Then, there's a loop which iterates `n - 2` times (the string length minus 2, as the loop starts at 1 and ends at `len(s) - 1`). Each iteration performs a constant amount of work by checking `s[i]` and possibly updating `t` and `ans`.
- Combining both steps, the total number of operations is proportional to `2n - 2`.

Since these are linear operations, the time complexity of the algorithm is `O(n)`.

Space Complexity

The space complexity is determined by the amount of extra storage the algorithm uses in relation to the input size.

- The algorithm uses a fixed number of integer variables `ans`, `t`, and `i`, which use a constant amount of space.
- There is no use of any data structure that grows with the input size.

Therefore, the space complexity is `O(1)`, as the amount of extra space used does not depend on the size of the input.