1573. Number of Ways to Split a String

Medium Math String Leetcode Link

Problem Description

The problem involves a binary string `s`, which consists only of '0's and '1's. The task is to find out how many different ways we can split this string into three non-empty substrings (`s1`, `s2`, and `s3`) such that the number of '1's is the same in each substring. If this condition cannot be met, we should return 0. If there are multiple ways to split the string that satisfy the condition, we should return the total count of such possibilities. Since the number of ways can be very large, it is only required to return the result modulo `10^9 + 7`.

Here's a more detailed explanation:

- If `s` is split into `s1`, `s2`, and `s3`, then `s1 + s2 + s3 = s`, which means they are consecutive parts of the original string.
- All three parts must have the same number of '1's, which also implies that the total number of '1's in `s` must be divisible by 3.
- If `s` doesn't contain any '1's, any split that partitions `s` into three parts would satisfy the condition since each part will have zero '1's.
- If the total number of '1's in `s` isn't divisible by 3, there is no way to split `s` to satisfy the given condition, so the result is 0.

Intuition

The solution is built on the understanding that to split the string into three parts with an equal number of '1's, the number of '1's must be divisible by 3.

- We start by counting the total number of '1's in the string. If the count is not divisible by 3, we can immediately return 0, since it's impossible to divide the '1's equally into three parts.
- If there are no '1's, then the binary string consists only of '0's. The string can then be split in any place between the '0's. The number of ways to choose two split points in a string of length `n` is `(n-1)*(n-2)/2`.
- If there are '1's in the string, we need to find the split points that give us the correct distribution of '1's. To do this, the following steps are taken:
  - We find the indices of the first and second occurrences of the `cnt`-th '1' (where `cnt` is the total number of '1's divided by 3), as well as the indices for the first and second occurrences of the 2*`cnt`-th '1'.
  - These indices help us identify the positions at which we can make our splits.
  - The number of ways to make the first split is the difference between the second and first occurrences of the `cnt`-th '1'.
  - The number of ways to make the second split is similarly the difference between the second and first occurrences of the 2*`cnt`-th '1'.
  - Multiplying these two numbers gives us the total number of ways to split the string to ensure all three parts have the same number of '1's.

The modulus operation is used to ensure the result stays within numerical bounds as per the problem's instructions.

Solution Approach

In the given solution, the algorithm begins with some pre-processing to calculate the total number of '1's present in the input string `s`. This uses the `sum` function and a generator expression to count '1's, `sum(c == '1' for c in s)`. The result is then divided by 3 with the `divmod` function, which returns a tuple containing the quotient and the remainder.

- If the remainder is not zero, it indicates that it's impossible to distribute '1's equally into three parts, and the function immediately returns `0`.
- If the quotient is zero (i.e., no '1's in the string), the solution uses the combination formula to calculate the number of ways to choose two positions out of `n-1` possible positions as split points. This is done using the formula `((n - 1) * (n - 2) // 2) % mod`, where `n` is the length of the string and `mod` is the required modulus `10^9 + 7`.

Next, the solution involves finding the exact points to split the string when '1's exist. For this purpose, the `find` function is defined, which iterates over the string and counts '1's until it reaches a target number (for example, the `cnt`-th '1' or 2*`cnt`-th '1'). It returns the index where this occurs.

- The `find` function is used four times to find two pairs of indices: `i1, i2`, which are the indices before and after the last '1' in the first third of '1's, and `j1, j2`, the corresponding indices for the second third of '1's.
- `i1` is obtained by looking for the `cnt`-th '1', while `i2` is obtained by searching for the occurrence of one more '1' after `i1` (i.e., `cnt + 1`). Similarly, `j1` is the index of the 2*`cnt`-th '1' and `j2` for one more '1' after `j1` (i.e., 2*`cnt` + 1).
- These indices mark the possible split points just before and after the identified '1's.

Once the split points are determined, the total number of ways to split `s` is the product of the number of ways to split at each pair of indices `i` and `j`, effectively `(i2 - i1) * (j2 - j1)`. The product is taken modulo `mod` to avoid large numbers and comply with the problem's requirement to output the result modulo `10^9 + 7`.

The solution effectively leverages basic counting principles and a single pass through the string to accomplish the task, ensuring a linear time complexity proportionate to the length of the string `s`. It uses very few additional data structures outside of basic counters and indices.

Example Walkthrough

Let's consider a small example to illustrate the solution approach with a binary string `s = "0011001"`.

1. First, we count the total number of `'1'`s in the string `s`. There are `3` ones in `0011001`.
2. We check if the total number of `'1'`s is divisible by 3. Since `3` is divisible by `3`, we can proceed.
3. We need to find the partition points in the string such that each partition contains exactly one `'1'`.
4. We start scanning the string to find the `1st '1'`, which is the `cnt`-th `'1'` (in our case, `1` as `3/3=1`), and we find it at index `2`.
5. We continue scanning to find the occurrence of the next `'1'` after the `cnt`-th `'1'`, which is at index `3`. This gives us our first potential split between indices `2` and `3`.
6. We repeat the process for finding the `2*cnt`-th `'1'` and the first occurrence after that. We find the next `'1'` at index `4` and the one after that at index `6`.
7. These indices give us the second potential split between indices `4` and `6`.

Now, let's calculate how many ways we can split `s` at these points:

- The first split can occur at either index `2` or `3`, so 2 ways (`i2 - i1` where `i1 = 2` and `i2 = 3`).
- The second split can occur at either index `4` or `6`, so 3 ways (`j2 - j1` where `j1 = 4` and `j2 = 6`).

To find the total number of ways to split the string to ensure all three parts have the same number of `'1'`s, we multiply the number of ways for each split.

- This gives us `2 * 3 = 6` ways to split the string `0011001` into three parts, each containing the same number of `'1'`s.

The result we obtained is the direct application of the algorithm described in the solution. The modulus operation is not necessary in this example because our final count is already within the bounds of `10^9 + 7`. However, in a solution implementation, the count would be taken modulo `10^9 + 7` as specified.

Python Solution

```python
class Solution:
    def numWays(self, s: str) -> int:
        def find_nth_occurrence(x):
            """Find the index of the nth occurrence of '1' in the string."""
            total_ones = 0
            # Enumerate over string characters and count ones
            for index, char in enumerate(s):
                total_ones += int(char == '1')
                # When the nth occurrence is reached, return the index
                if total_ones == x:
                    return index
            return None  # If the nth occurrence doesn't exist

        # Count the total number of '1's in the string and check if it can be divided into 3 parts
        ones_count, remainder = divmod(sum(char == '1' for char in s), 3)

        # If it's not divisible by 3, there are no ways to split, return 0
        if remainder:
            return 0

        n = len(s)  # Length of the input string
        # A modulus constant for the result as per LeetCode's requirement
        mod = 10**9 + 7

        # When there are no '1's, we can choose 2 points to split the '0's into 3 parts
        if ones_count == 0:
            # Computing combination n-1 choose 2
            return ((n - 1) * (n - 2) // 2) % mod

        # Find the indices for splitting
        # The first split is after the 'ones_count'th '1'
        first_split_index_start = find_nth_occurrence(ones_count)
        # The second split is after the first '1' that follows the first split
        first_split_index_end = find_nth_occurrence(ones_count + 1)

        # Similarly for the second split
        second_split_index_start = find_nth_occurrence(ones_count * 2)
        second_split_index_end = find_nth_occurrence(ones_count * 2 + 1)

        # Calculate the number of ways to split for the first and second split
        # Multiply them and return the result modulo 10^9 + 7
        return (first_split_index_end - first_split_index_start) * (second_split_index_end - second_split_index_start) % mod

# Example usage:
sol = Solution()
result = sol.numWays("10101")  # Should return 4 as there are four ways to split "10101" into three parts with equal number of '1's
print(result)  # Output: 4

```

Java Solution

```java
class Solution {
    private String binaryString; // Renamed s to binaryString for clarity

    // Method to count the number of ways to split the given string in three parts with an equal number of '1's
    public int numWays(String binaryString) {
        this.binaryString = binaryString; // Initializing the class-level variable
        int onesCount = 0; // Counter for the number of '1's in the string
        int stringLength = binaryString.length(); // Store the length of the string

        // Count the number of '1's in the string
        for (int i = 0; i < stringLength; ++i) {
            if (binaryString.charAt(i) == '1') {
                ++onesCount;
            }
        }

        int remainder = onesCount % 3; // Calculate remainder to check if onesCount is divisible by 3
        if (remainder != 0) {
            // If not divisible by three, return 0 as it's impossible to split the string properly
            return 0;
        }
        final int mod = (int) 1e9 + 7; // Modulus value for the result

        // If the string contains no '1's, calculate the number of ways to split the zeros
        if (onesCount == 0) {
            // Number of subarrays is a combination: Choose 2 from stringLength - 1 and take mod
            return (int) (((stringLength - 1L) * (stringLength - 2) / 2) % mod);
        }

        onesCount /= 3; // Divide the count of '1's by 3 to find the size of each part
        // Find positions around the first third
        long firstCutStart = findCutPosition(onesCount);
        long firstCutEnd = findCutPosition(onesCount + 1);
        // Find positions around the second third
        long secondCutStart = findCutPosition(onesCount * 2);
        long secondCutEnd = findCutPosition(onesCount * 2 + 1);
        // Calculate the number of ways to make the cuts and take mod
        return (int) ((firstCutEnd - firstCutStart) * (secondCutEnd - secondCutStart) % mod);
    }

    // Helper method to find the cut positions in the binary string
    private int findCutPosition(int targetOnesCount) {
        int tempCounter = 0; // Temporary counter to track the number of '1's
        // Look for the position in the string that corresponds to the target count of '1's
        for (int i = 0;; ++i) {
            tempCounter += binaryString.charAt(i) == '1' ? 1 : 0;
            if (tempCounter == targetOnesCount) {
                // Once found, return the index of the string at that count
                return i;
            }
        }
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int numWays(string s) {
        // Count the number of '1's in the string.
        int oneCount = 0;
        for (char c : s) {
            oneCount += (c == '1');
        }

        // If oneCount is not divisible by 3, there's no way to split.
        if (oneCount % 3 != 0) {
            return 0;
        }

        // Define modulo for the result as required by the problem.
        const int MOD = 1e9 + 7;
        int length = s.size();

        // If there are no '1's in the string, return the number of ways to choose
        // the points to split the string in 3 parts, avoiding permutations.
        if (oneCount == 0) {
            return ((long long)(length - 1) * (long long)(length - 2) / 2) % MOD;
        }

        // Adjust oneCount to be one third of the original oneCount
        oneCount /= 3;

        // Helper lambda function to find the index of the start of a particular group of '1's.
        auto findIndexOfGroupStart = [&](int groupCount) -> int {
            int count = 0;
            for (int i = 0; ; ++i) {
                count += (s[i] == '1');
                if (count == groupCount) {
                    return i;
                }
            }
        };

        // Find the indices where the first and second splits should occur.
        int firstSplitStart = findIndexOfGroupStart(oneCount);
        int firstSplitEnd = findIndexOfGroupStart(oneCount + 1);
        int secondSplitStart = findIndexOfGroupStart(oneCount * 2);
        int secondSplitEnd = findIndexOfGroupStart(oneCount * 2 + 1);

        // Calculate the number of ways to make the splits and return the result module MOD.
        return ((long long)(firstSplitEnd - firstSplitStart) * (long long)(secondSplitEnd - secondSplitStart)) % MOD;
    }
};

```

Typescript Solution

```typescript
// Global variable to define modulo for the result as required by the problem
const MOD: number = 1e9 + 7;

// Function to count the number of '1's in the string
function countOnes(s: string): number {
    let oneCount = 0;
    for (let c of s) {
        oneCount += (c === '1') ? 1 : 0;
    }
    return oneCount;
}

// Function to find the index of the start of a particular group of '1's
function findIndexOfGroupStart(s: string, groupCount: number): number {
    let count = 0;
    for (let i = 0; i < s.length; i++) {
        if (s[i] === '1') count++;
        if (count === groupCount) {
            return i;
        }
    }
    return -1; // This condition should not happen due to input constraints
}

// Function that calculates the number of ways the input string s can be split into three parts with equal number of '1's
function numWays(s: string): number {
    let oneCount = countOnes(s);
    let length = s.length;

    // If oneCount is not divisible by 3, there's no way to split the string
    if (oneCount % 3 !== 0) {
        return 0;
    }

    // If there are no '1's in the string, return the number of ways to choose
    // the points to split the string in 3 parts, avoiding permutations.
    if (oneCount === 0) {
        // Using modular arithmetic for calculating binomial coefficient (n - 1) choose 2
        return (((length - 1) * (length - 2) / 2) % MOD);
    }

    // Adjust oneCount to be one third of the original oneCount
    oneCount /= 3;

    // Find the indices where the first and second splits should occur
    let firstSplitStart = findIndexOfGroupStart(s, oneCount);
    let firstSplitEnd = findIndexOfGroupStart(s, oneCount + 1);
    let secondSplitStart = findIndexOfGroupStart(s, oneCount * 2);
    let secondSplitEnd = findIndexOfGroupStart(s, oneCount * 2 + 1);

    // Calculate the number of ways to make the first and the second split.
    // Multiply the number of zeroes between the adjacent groups of '1', and take the result module MOD
    return ((firstSplitEnd - firstSplitStart) * (secondSplitEnd - secondSplitStart)) % MOD;
}

// Example usage:
// const s: string = "10101";
// console.log(numWays(s)); // The output would be 4

```

Time and Space Complexity

Time Complexity

The given code snippet includes three significant calculations: counting the number of 1's in the string, finding the indices where certain counts of 1's occur, and calculating the total number of ways to split the string.

1. Counting the number of 1's: This is done by iterating over each character in the string once. This operation takes `O(n)` time where `n` is the length of the string `s`.

2. Finding the indices with the help of the `find` function: This function is called four times. Each call to the `find` method iterates over the entire string in the worst-case scenario, which makes it `O(n)` for each call. Therefore, for four calls, the time complexity associated with the `find` function is `O(4 * n)` which simplifies to `O(n)`.

3. Calculating combinations for zero counts of 1's: The calculation `((n - 1) * (n - 2) // 2) % mod` is executed in constant time `O(1)`, as it doesn't depend on the size of the input string beyond the length calculation.

Hence, the overall time complexity of the code is `O(n)` for scanning the string to count the number of 1's plus `O(n)` for the `find` operations, which simplifies to `O(n)`.

Space Complexity

1. `cnt` and `m`: These are constant-size integers, which occupy `O(1)` space.

2. Counting 1's: The sum comprehension iterates over the string and counts the number of '1's, which does not require additional space, so it is `O(1)`.

3. Variables `i1`, `i2`, `j1`, `j2`, and `n`: They are also constant-size integers, with no additional space dependent on the input size, so this is `O(1)`.

4. The function `find` uses `i` and `t` which again are constant-size integers, hence `O(1)` space.

As there are no data structures used that scale with the size of the input, the overall space complexity of the code is `O(1)`, which represents constant space usage.