1513. Number of Substrings With Only 1s

Medium Math String Leetcode Link

Problem Description

The problem requires us to find and count the number of substrings within a given binary string `s` that consist entirely of the character '1'. For example, in the string "0110111", there are several such substrings: "1", "11", "111", "11", "1", and each of them should be counted. The challenge here is to do this efficiently and to deal with potentially very large numbers, the counts need to be returned modulo `10^9 + 7`, which is a large prime number commonly used to prevent integer overflow issues in competitive programming.

Intuition

The intuition behind the solution is based on the observation that for a sequence of '1's of length `n`, there are exactly `n * (n + 1) / 2` substrings that can be formed, all of which consist solely of '1's. This is because each additional '1' potentially adds as many new substrings as its position in the sequence (first '1' adds 1 substring, second '1' adds 2 substrings, and so on). So, we are essentially counting the lengths of continuous sequences of '1's and calculating the number of substrings for each sequence.

The solution involves iterating through the string and using a counter variable `cnt` to keep track of the length of a consecutive sequence of '1's. When we encounter a '1', we increment `cnt`. If we encounter a '0', we reset `cnt` to zero, because it breaks the sequence of '1's. After each step, regardless of whether we're continuing a sequence or ending one, we add the current value of `cnt` to `ans`, which accumulates the total number of substrings of '1's we've seen so far.

Finally, since the answer could be a very large number, we return the total count modulo `10^9 + 7` as instructed by the problem description.

Solution Approach

The implementation of the solution is relatively straightforward with no need for complex data structures; the only variable that holds state during iteration is `cnt`, which counts the length of consecutive '1's. With every step forward in the string, the algorithm performs the following actions:

1. Check the current character (`c`) of the string.
2. If `c` is `'1'`, increment the `cnt` variable: this increases the length of the current sequence of '1's by one.
3. If `c` is not `'1'` (in this case, it could only be `'0'` since it's a binary string), reset `cnt` to zero. This indicates the end of the current sequence of '1's and resets the count for the next sequence.
4. After updating `cnt`, add its current value to `ans`. This step accumulates the total number of valid substrings formed by the sequences of '1's encountered so far.

This process works because each time a '1' is encountered and `cnt` is incremented, it effectively counts all the new substrings that could end with this '1'. If you have a sequence of length `n`, then adding one more '1' to the sequence adds `n + 1` substrings: all the previous substrings plus a new substring that includes the entire sequence.

The use of the modulo operation `% (10**9 + 7)` ensures that we handle the possibility of very large outputs, preventing integer overflow and keeping the result within the limits of the problem's constraints.

No additional data structures are required, and the pattern used in this algorithm is commonly known as a single pass or linear scan, as it requires only one iteration through the input string to compute the result.

Example Walkthrough

Let's consider a small example binary string `s = "11011"`. Here's how we would apply the solution approach to this input:

1. Initialize `cnt = 0` (to count the consecutive '1's) and `ans = 0` (to accumulate the total number of substrings).
2. Start iterating over the string `s` from left to right.
   - Character `s[0]` is '1':
     - Increment `cnt` to `1`. Current substrings are: "1".
     - Add `cnt` to `ans`: `ans += 1`, hence `ans = 0 + 1 = 1`.
   - Character `s[1]` is '1':
     - Increment `cnt` to `2`. Current substrings are: "1", "11".
     - Add `cnt` to `ans`: `ans += 2`, hence `ans = 1 + 2 = 3`.
3. The next character `s[2]` is '0', which interrupts the sequence of '1's:
   - Reset `cnt` to `0`.
   - `ans` remains unchanged.
4. Continue iteration.
   - Character `s[3]` is '1':
     - Increment `cnt` to `1`. Current substrings are: "1".
     - Add `cnt` to `ans`: `ans += 1`, hence `ans = 3 + 1 = 4`.
   - Character `s[4]` is '1':
     - Increment `cnt` to `2`. Current substrings are: "1", "11".
     - Add `cnt` to `ans`: `ans += 2`, hence `ans = 4 + 2 = 6`.
5. At this point, we have finished iterating over the entire string. The total count of substrings consisting only of '1's is stored in `ans`, which equals `6`.
6. Return `ans % (10^9 + 7)` to adhere to the problem constraints. Since `6` is well below the modulo, the final result is simply `6`.

As we can see from the example, the algorithm correctly counts the individual substrings of '1's and sums them up. It counts "11" at the beginning as 2 substrings, resets the count after the '0', and then counts "1" and "11" again, adding up to a total of 6 substrings consisting of '1's.

Python Solution

```python
class Solution:
    def numSub(self, s: str) -> int:
        # Initialize the answer and a temporary counter for consecutive "1"s
        total_substrings = consecutive_ones_count = 0

        # Iterate through each character in the string
        for char in s:
            # If the character is "1", increment the consecutive "1"s counter
            if char == "1":
                consecutive_ones_count += 1
                # Add the current count of consecutive "1"s to the total substrings.
                # Each time we encounter a "1", it forms a substring with all the consecutive "1"s before it.
                total_substrings += consecutive_ones_count
            # If the character is not "1", reset the consecutive "1"s counter to 0
            else:
                consecutive_ones_count = 0

        # Return the total number of substrings, modulo 10^9 + 7
        return total_substrings % (10**9 + 7)

```

Java Solution

```java
class Solution {
    // This method counts the number of substrings that contains only the character '1'
    public int numSub(String s) {
        final int MODULO = (int) 1e9 + 7; // Define a constant for modulo operation
        int countSubstrings = 0; // Store the count of valid substrings
        int consecutiveOnes = 0; // Counter for consecutive '1's

        // Iterate through the string character by character
        for (int i = 0; i < s.length(); ++i) {
            // If the current character is '1', increment the consecutive ones counter
            // Otherwise, reset it to zero since we're only interested in consecutive '1's
            consecutiveOnes = s.charAt(i) == '1' ? consecutiveOnes + 1 : 0;

            // Add the current count of consecutive '1's to countSubstrings
            // This works because for each new '1' character, it forms a new substring
            // ending at this character that has not been counted yet
            countSubstrings = (countSubstrings + consecutiveOnes) % MODULO;
        }

        // Return the total number of substrings found, modulo the defined constant
        return countSubstrings;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to count the number of substrings containing only '1's
    int numSub(string s) {
        long long answer = 0; // Initialize the answer to 0
        int count = 0; // Initialize a counter for consecutive '1's
        const int MODULUS = 1e9 + 7; // Define the modulus value for preventing integer overflow

        // Iterate over each character in the input string
        for (char& character : s) {
            // If the current character is '1', increase the count of consecutive '1's
            // Otherwise, reset the count to 0
            count = character == '1' ? count + 1 : 0;

            // Add the current count to the answer. This works because for each '1' found,
            // it contributes to that many substrings ending at that point (e.g., "111" has
            // substrings "1", "11", and "111" ending at the last character).
            // Apply modulus operation to keep the number within bounds and handle overflow
            answer = (answer + count) % MODULUS;
        }

        // Return the final count of substrings
        return static_cast<int>(answer); // Cast the long long answer to int before returning
    }
};

```

Typescript Solution

```typescript
function numSub(sequence: string): number {
    // Define the modulus value to keep the result within integer limits
    const modulus = 10 ** 9 + 7;

    // Initialize the answer to be returned
    let answer = 0;

    // Counter for the consecutive '1's
    let count = 0;

    // Iterate through each character in the string
    for (const char of sequence) {
        // If the character is '1', increase count, otherwise reset count to 0
        count = (char === '1') ? count + 1 : 0;

        // Add the current count to the answer and apply modulus to prevent overflow
        answer = (answer + count) % modulus;
    }

    // Return the computed answer
    return answer;
}

```

Time and Space Complexity

The code provided counts the number of substrings of contiguous '1's in a given string `s`. The time complexity and space complexity of the algorithm are analyzed below.

Time Complexity

The algorithm goes through each character of the input string `s` exactly once. During this process, it performs constant time operations such as checking if a character is '1' or '0', incrementing the `cnt` variable, and adding the value of `cnt` to `ans`. Because these operations do not depend on the size of the string and are repeated for each character, the time complexity is directly proportional to the length of the string. Therefore, the **time complexity** is `O(n)`, where `n` is the length of the input string `s`.

Space Complexity

The space used by the algorithm includes a fixed number of integer variables `ans` and `cnt`, which do not scale with the size of the input string. As a result, the algorithm uses a constant amount of additional space, resulting in a **space complexity** of `O(1)`, indicating that it is independent of the input size.