2380. Time Needed to Rearrange a Binary String

Medium String Dynamic Programming Simulation

Problem Description

The problem gives us a binary string `s`, which consists of '0's and '1's. We are then introduced to a procedure that takes place every second where every occurrence of the substring "01" is simultaneously replaced by "10". This process continues to happen every second until there are no more occurrences of "01" left in the string. Our task is to determine how many seconds it will take until the process can no longer continue, meaning the string has no occurrences of "01".

Intuition

To solve this problem, one could think about how the string evolves over time. Consider a "01" pattern. When you replace it with "10", the '0' effectively moves to the right. This process continues until all '0's have no '1's to their left. In other words, we want all the '0's in the string to be on the left side and all the '1's to be on the right side, and we count how many seconds it takes for this to happen.

To find the solution, we don't need to simulate the entire process. Instead, we keep track of how many '0's we have passed as we iterate through the string (`cnt`). When we encounter a '1', we know that if there is any '0' to the left of this '1' (which means `cnt` would be more than 0), we'll have a "01" pattern.

The key insight is that for each '1' found, if there was at least one '0' before it, we need at least one second for the leftmost '0' to move past this '1'. However, if we encounter another '1' while still having '0's in our count, we'll need an additional second. Essentially, we continue incrementing our answer until there are no more '0's to move across '1's.

The variable `ans` keeps track of the maximum number of seconds needed so far. We either increment `ans` by 1 or update it to `cnt` if `cnt` is greater (which means we encountered a '1' with many '0's before it, increasing the required time). Once we've gone through the whole string, `ans` will contain the number of seconds needed to complete the process.

Solution Approach

The solution uses a simple linear scan of the binary string, and it's based on the observation that every second, a '0' can move past a '1' if and only if there is a '0' to its immediate left.

Here's a detailed walk-through of the solution:

1. Initialize `ans` to 0, which is used to store the maximum number of seconds required to get all '0's to the left side of all '1's in the string.
2. Initialize `cnt` to 0, which is used to keep track of the number of '0's encountered while iterating through the string from left to right.
3. Iterate through each character `c` in the binary string `s`:
   - If `c` is '0', this means we have one more '0' that might need to move rightward, so we increment `cnt`.
   - If `c` is '1' and `cnt` is greater than 0, this indicates there are '0's that need to move past this '1'. Therefore, we calculate the number of seconds required. This is the maximum of the current value of `ans` incremented by 1 (as we need at least one more second for a '0' to move past this '1') and `cnt` (which represents the bulk move of all the '0's we have encountered so far). We update `ans` to this calculated value.
4. The loop continues until the end of the string. By the end of the loop, `ans` will hold the total number of seconds needed, as it captures the most time-consuming scenario of moving all '0's to the left of all '1's.
5. Return `ans` as the result.

This approach doesn't require complex data structures or algorithms, simply iterating through the string and keeping track of two integer variables (`ans` and `cnt`). It efficiently arrives at the solution by focusing on when '1's are encountered and how many '0's are to their left—a fundamental aspect of how the process evolves over time. The pattern used here is essentially a greedy approach, optimizing for each '1' found in the string.

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider the binary string `s = "001011"`.

1. Initialize `ans` to 0 and `cnt` to 0.
2. Start iterating over the string from left to right:
   - First character is '0': increment `cnt` to 1.
   - Second character is '0': increment `cnt` to 2.
   - Third character is '1': `cnt` is greater than 0, so we have '0's that need to move past this '1'. We determine the temporary number of seconds which would be the maximum of `ans+1` (0+1=1) and `cnt` (2). So we update `ans` to 2.
   - Fourth character is '0': increment `cnt` to 3.
   - Fifth character is '1': again `cnt` is greater than 0, indicating '0's need to move past this '1'. Calculate the maximum of `ans+1` (2+1=3) and `cnt` (3). `ans` remains 3.
   - Sixth character is '1': `cnt` is still greater than 0, so we perform the same calculation: maximum of `ans+1` (3+1=4) and `cnt` (3). `ans` is updated to 4.
3. The iteration completes with a final `ans` of 4, which is the total number of seconds needed for the string to have no occurrences of "01".

Following the steps outlined in the solution approach, we've determined it will take 4 seconds for the string "001011" to become "000111", at which point the process cannot continue as there are no more occurrences of "01" left in the string.

Solution Implementation

Python

```python
class Solution:
    def secondsToRemoveOccurrences(self, s: str) -> int:
        # Initialize a variable to keep track of the number of seconds.
        seconds = 0
        # Initialize a counter for the number of zeros seen so far.
        zero_count = 0

        # Iterate over each character in the string.
        for char in s:
            if char == '0':
                # Increment zero count on seeing a '0'.
                zero_count += 1
            elif zero_count:
                # If a '1' is encountered and there was at least one '0' before it,
                # increment seconds needed. It either takes one more second than the
                # previous required seconds, or the number of observed zeros, whichever is larger.
                # This is because you need at least as many seconds as the number of zeros
                # you'll need to move past each '1' once.
                seconds = max(seconds + 1, zero_count)

        # Return the total number of seconds needed to get all '1's to the right of all '0's.
        return seconds
```

Java

```java
class Solution {
    public int secondsToRemoveOccurrences(String s) {
        int secondsRequired = 0; // initializes the count of seconds required to remove all occurrences
        int countZeros = 0; // initializes the count of '0's encountered that need to be moved

        // Loop through each character in the string 's'
        for (char character : s.toCharArray()) {

            if (character == '0') {
                // Increment the count of '0's when a '0' is encountered
                ++countZeros;
            } else if (countZeros > 0) {
                // If a '1' is encountered and there are '0's that need to be moved,
                // we increment the seconds required. The logic behind this is that for
                // each '1' we encounter after some '0's, we need at least one move to start
                // moving all those '0's past this '1'. This essentially tracks the batches of movements required.

                // The max function ensures that if we have a consecutive batch of '0's followed by '1's larger
                // than any previous batch, we use that larger value as the seconds required.
                secondsRequired = Math.max(secondsRequired + 1, countZeros);
            }
        }

        // Return the number of seconds required to have no '01' occurrences in the string
        return secondsRequired;
    }
}
```

C++

```cpp
class Solution {
public:
    int secondsToRemoveOccurrences(string s) {
        int maxSeconds = 0;  // This will keep track of the total seconds needed to remove all occurrences.
        int zeroCount = 0;   // This will count the number of zeros we have seen so far.

        // Iterate over each character in the string.
        for (char currentChar : s) {
            // If the current character is '0', increment the zeroCount.
            if (currentChar == '0') {
                ++zeroCount;
            }
            // If it is '1' and we have seen at least one '0' before it,
            // we need to perform a swap operation. This means that we need
            // at least one second for each '1' after the first '0'.
            else if (zeroCount) {
                // The number of seconds required can be either one more than the number
                // of seconds calculated so far or the number of zeros seen, whichever is larger.
                // This accounts for the fact that we can move each '1' past all '0's seen so far
                // by continuous swaps which take one second each.
                maxSeconds = max(maxSeconds + 1, zeroCount);
            }
        }

        // The value of maxSeconds is the total seconds required to remove all
        // "01" occurrences in the string by swapping adjacent characters.
        return maxSeconds;
    }
};
```

TypeScript

```typescript
// Define secondsToRemoveOccurrences as a global function that takes a string and returns a number.
function secondsToRemoveOccurrences(s: string): number {
    let maxSeconds = 0;  // This holds the total seconds needed to remove all "01" sequences.
    let zeroCount = 0;   // This counts the number of '0's encountered up to the current position.

    // Loop through each character in the string.
    for (let i = 0; i < s.length; i++) {
        // If the current character is '0', increment zeroCount.
        if (s[i] === '0') {
            zeroCount++;
        }
        // If the current character is '1' and we have seen '0's before it.
        else if (zeroCount > 0) {
            // Increment maxSeconds or set it to zeroCount, whichever is larger.
            // This handles the need to move '1' past all '0's seen so far, one second per swap.
            maxSeconds = Math.max(maxSeconds + 1, zeroCount);
        }
    }

    // Return the calculated maximum number of seconds to remove all "01" occurrences in the string.
    return maxSeconds;
}

// Example Usage
// const sequence: string = "001011";
// console.log(secondsToRemoveOccurrences(sequence));  // Output will be the number of seconds required.
```

```python
class Solution:
    def secondsToRemoveOccurrences(self, s: str) -> int:
        # Initialize a variable to keep track of the number of seconds.
        seconds = 0
        # Initialize a counter for the number of zeros seen so far.
        zero_count = 0

        # Iterate over each character in the string.
        for char in s:
            if char == '0':
                # Increment zero count on seeing a '0'.
                zero_count += 1
            elif zero_count:
                # If a '1' is encountered and there was at least one '0' before it,
                # increment seconds needed. It either takes one more second than the
                # previous required seconds, or the number of observed zeros, whichever is larger.
                # This is because you need at least as many seconds as the number of zeros
                # you'll need to move past each '1' once.
                seconds = max(seconds + 1, zero_count)

        # Return the total number of seconds needed to get all '1's to the right of all '0's.
        return seconds
```

Time and Space Complexity

The given Python code defines a function `secondsToRemoveOccurrences` which takes a string `s` as its argument and calculates the number of seconds needed to remove all occurrences of the pattern "01" by repeatedly replacing it with "10" until no more occurrences are left.

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the input string `s`. We can deduce this because the function contains a single loop that iterates through each character of the string exactly once. Inside the loop, it performs constant-time operations, including comparison, arithmetic operations, and assignment. There are no nested loops or recursive calls that would increase the complexity. Therefore, the time taken by the algorithm grows linearly with the size of the input string.

Space Complexity

The space complexity of the code is `O(1)`. The function uses a fixed amount of extra space, regardless of the input size. The variables `ans` and `cnt` are used to keep track of the current state while iterating over the string, but no additional space that grows with the input size is allocated. The input string `s` is not modified, and no other data structures are used. Hence, the amount of memory used by the function remains constant, irrespective of the input size.