# 1869. Longer Contiguous Segments of Ones than Zeros

Easy String

Leetcode Link

## Problem Description

The problem presents us with a binary string `s`, which consists only of `0`s and `1`s. Our goal is to determine whether the longest sequence of consecutive `1`s is longer than the longest sequence of consecutive `0`s in the binary string. If the longest sequence of `1`s is longer, we should return `true`; otherwise, we return `false`. A sequence of consecutive `1`s or `0`s means they are directly next to each other without being interrupted by the other digit.

For example, let's take a binary string `s = "110100010"`. Here, the longest segment of `1`s has length `2`, and the longest segment of `0`s has a length of `3`. So in this case, the function should return `false` because the segment of `0`s is longer than the segment of `1`s.

If there are no `0`s in the string, the length of the longest segment of `0`s is `0`, and similarly, if there are no `1`s, the length of the longest segment of `1`s is `0`.

## Intuition

The algorithm needs to go through the string `s` and track the length of the current segment of consecutive `1`s and `0`s, as well as the length of the longest segments found so far. The idea is to iterate through the string once and, for each character:

- If the character is a `0`, we increase the temporary count of `0`s (`t0`) by 1 and reset the temporary count for `1`s (`t1`) to 0 because the sequence of `1`s is interrupted.
- Similarly, if the character is a `1`, we do the opposite: increase the count for `1`s and reset the count for `0`s.

While performing these operations, we also keep track of the maximum lengths seen so far for both `0`s and `1`s. This is done by comparing the current temporary counts (`t0` and `t1`) with the max counts (`n0` and `n1`) respectively and updating the max counts if the current is greater.

At the end of the iteration, we simply compare the longest segment of `1`s to the longest segment of `0`s and return `true` if the segment of `1`s is longer; otherwise, we return `false`.

This solution is efficient because it only requires a single pass through the string, making it a linear time algorithm with O(n) complexity, where n is the length of the string.

## Solution Approach

The solution to the "Check if Binary String Has at Most One Segment of Ones" problem relies on an approach that allows us to find the longest contiguous segments of `1`s and `0`s in a single pass through the string.

Here is a step-by-step walkthrough of the implementation from the provided solution code:

1. Initialize four variables: `n0` and `n1` are set to 0 to keep track of the maximum lengths of the segments of `0`s and `1`s respectively; `t0` and `t1` are set to 0 to keep the count of the current length of the segments of `0`s and `1`s.
2. Iterate through each character `c` in the binary string `s`:
   - If `c` is `'0'`, increment the count of `t0` because we are in a segment of `0`s, and reset `t1` to 0 as we're no longer in a segment of `1`s.
   - If `c` is `'1'`, increment the count of `t1` because we are in a segment of `1`s, and reset `t0` to 0 as we're no longer in a segment of `0`s.
3. After every iteration (for each character `c`), refresh the maximum values `n0` and `n1` by comparing them with `t0` and `t1`. This ensures that after processing the complete string, `n0` and `n1` hold the lengths of the longest segments of `0`s and `1`s, respectively.
   - Update `n0` by evaluating `n0 = max(n0, t0)`.
   - Update `n1` by evaluating `n1 = max(n1, t1)`.
4. Finally, once the iteration is finished, return the result of comparing `n1` and `n0` by returning `True` if `n1 > n0` and `False` otherwise. This comparison dictates whether the longest contiguous segment of `1`s is strictly longer than the longest contiguous segment of `0`s.

No additional data structures or complex patterns are needed for the implementation. The efficiency lies in the simplicity of using counting variables and updating maximums, which only employ basic operations.

The algorithm, thus, performs in O(n) time complexity, where n is the length of the binary string, because it requires just one scan through the string. The space complexity is O(1), as the amount of extra space used does not grow with the size of the input.

### Example Walkthrough

Let us consider a small example with the binary string `s = "0110011"`. When this string is passed through the algorithm described in the solution approach, here's a step-by-step explanation of what happens:

1. Initialize variables: `n0 = 0`, `n1 = 0` to keep track of the maximum lengths of `0`s and `1`s, and `t0 = 0`, `t1 = 0` to keep track of the current lengths.
2. Iteration starts:
   - `c = '0'`: `t0` becomes 1, `t1` is reset to 0. We compare `n0` with `t0`, now `n0=1` and `n1=0`.
   - `c = '1'`: `t1` becomes 1, `t0` is reset to 0. Since `n1` (which is 0) is less than `t1` (which is 1), we update `n1=1`. Now `n0=1` and `n1=1`.
   - `c = '1'`: `t1` increases to 2, `t0` remains 0. Update `n1` to 2, as it is greater than the current `n1`. Now `n0=1` and `n1=2`.
   - `c = '0'`: `t0` becomes 1, `t1` is reset to 0, do not update `n0` or `n1` as current values of `t0` and `t1` are not greater than `n0` and `n1`.
   - `c = '0'`: `t0` increases to 2, `t1` remains 0. Update `n0` to 2, as it is equal to the current `t0`. Now `n0=2` and `n1=2`.
   - `c = '1'`: `t1` becomes 1, `t0` is reset to 0. Do not update `n0` or `n1`.
   - `c = '1'`: `t1` increases to 2, `t0` remains 0. Do not update `n1` as `t1` equals `n1`. Final values are `n0=2` and `n1=2`.
3. Once the iteration is over, we compare `n1` and `n0`. Since `n1` is not greater than `n0`, we return `False`.

No segment of `1`s is strictly longer than the longest segment of `0`s in the binary string `s = "0110011"`, so our algorithm correctly returns `False` based on the provided method. The complexity remains linear since we only went through the string a single time, and the space complexity is constant as we only kept track of a fixed number of variables.

## Python Solution

```python
class Solution:
    def checkZeroOnes(self, s: str) -> bool:
        # Initialize variables to keep track of the longest consecutive sequence of '0's and '1's
        max_zeros = max_ones = 0
        current_zeros = current_ones = 0

        # Iterate over the characters in the string
        for char in s:
            if char == '0':
                # Increase the sequence count of '0's, reset the sequence of '1's
                current_zeros += 1
                current_ones = 0
            else:
                # Do the opposite when we encounter '1's
                current_zeros = 0
                current_ones += 1

            # Update the maximum sequence lengths for '0's and '1's
            max_zeros = max(max_zeros, current_zeros)
            max_ones = max(max_ones, current_ones)

        # Check if the maximum sequence of '1's is greater than that of '0's
        return max_ones > max_zeros
```

## Java Solution

```java
class Solution {
    public boolean checkZeroOnes(String s) {
        int longestZeroSeq = 0; // Variable to track the length of the longest contiguous sequence of '0's
        int longestOneSeq = 0; // Variable to track the length of the longest contiguous sequence of '1's
        int currentZeroSeq = 0; // Variable to track the length of the current contiguous sequence of '0's
        int currentOneSeq = 0; // Variable to track the length of the current contiguous sequence of '1's

        // Iterate over each character in the string
        for (int i = 0; i < s.length(); ++i) {
            if (s.charAt(i) == '0') { // If the current character is '0'
                currentZeroSeq++; // Increase the length of the current '0' sequence
                currentOneSeq = 0; // Reset the length of the '1' sequence
            } else { // If the current character is '1'
                currentOneSeq++; // Increase the length of the current '1' sequence
                currentZeroSeq = 0; // Reset the length of the '0' sequence
            }
            // Update the longest sequence lengths if the current sequences are longer
            longestZeroSeq = Math.max(longestZeroSeq, currentZeroSeq);
            longestOneSeq = Math.max(longestOneSeq, currentOneSeq);
        }

        // After going through the string, check if the longest sequence of '1's is strictly greater than '0's
        return longestOneSeq > longestZeroSeq;
    }
}
```

## C++ Solution

```cpp
class Solution {
public:
    // Function to check if the longest continuous sequence of 1s is longer than the longest continuous sequence of 0s
    bool checkZeroOnes(string s) {
        int longestZero = 0; // Stores the length of the longest sequence of '0's
        int longestOne = 0;  // Stores the length of the longest sequence of '1's
        int currentZero = 0; // Tracks the current sequence length of '0's
        int currentOne = 0;  // Tracks the current sequence length of '1's

        // Loop through the input string character by character
        for (char c : s) {
            if (c == '0') {
                // If the current character is '0', increase the count of the current sequence of '0's
                ++currentZero;
                // Reset the current sequence of '1's since it's interrupted by a '0'
                currentOne = 0;
            } else { // c is '1'
                // If the current character is '1', increase the count of the current sequence of '1's
                ++currentOne;
                // Reset the current sequence of '0's since it's interrupted by a '1'
                currentZero = 0;
            }

            // Update the longest sequence length of '0's if necessary
            longestZero = max(longestZero, currentZero);
            // Update the longest sequence length of '1's if necessary
            longestOne = max(longestOne, currentOne);
        }

        // Return true if the longest sequence of '1's is greater than the longest sequence of '0's
        return longestOne > longestZero;
    }
};
```

## Typescript Solution

```typescript
/**
 * Determines if the longest contiguous segment of 1s is longer than the
 * longest contiguous segment of 0s in the binary string.
 *
 * @param {string} s - A binary string containing only '0's and '1's.
 * @return {boolean} - True if the longest segment of 1s is longer than
 *                     the longest segment of 0s; otherwise, false.
 */
const checkZeroOnes = (s: string): boolean => {
    let maxZeros: number = 0;   // Tracks the length of the longest segment of 0s.
    let maxOnes: number = 0;    // Tracks the length of the longest segment of 1s.
    let currentZeros: number = 0; // Tracks the length of the current segment of 0s.
    let currentOnes: number = 0;  // Tracks the length of the current segment of 1s.

    for (let char of s) {
        if (char === '0') {
            currentZeros++;   // Increment the count of contiguous 0s.
            currentOnes = 0;  // Reset the count of contiguous 1s.
        } else {
            currentOnes++;    // Increment the count of contiguous 1s.
            currentZeros = 0;  // Reset the count of contiguous 0s.
        }

        // Update the maximum counts if the current counts are greater.
        maxZeros = Math.max(maxZeros, currentZeros);
        maxOnes = Math.max(maxOnes, currentOnes);
    }
    // Return true if longest segment of 1s is longer than longest segment of 0s.
    return maxOnes > maxZeros;
};

// The function can be used as follows:
// const result: boolean = checkZeroOnes("11001");
// console.log(result); // Output would be true if the longest segment of 1s is longer than the longest segment of 0s.
```

## Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the input string `s`. This is because the code consists of a single for-loop that iterates over each character in the string exactly once.

During each iteration, the code does a constant amount of work: updating counters `t0` and `t1`, resetting one counter depending on the character, and calculating the maximum length of consecutive '0's and '1's seen so far using `max()`. Since all these operations are constant time, the loop represents a linear time complexity relative to the length of the string.

The space complexity of the code is `O(1)`. The amount of extra space used by the algorithm does not depend on the size of the input string. The variables `n0`, `n1`, `t0`, and `t1` use a fixed amount of space. No additional space that grows with input size is required. Therefore, the space complexity is constant.