1784. Check if Binary String Has at Most One Segment of Ones

Easy String

Leetcode Link

Problem Description

In the given problem, we are provided with a binary string `s`. A binary string is a string that contains only the characters '0' or '1'. Additionally, the string `s` does not have any leading zeros, which means it does not start with the character '0'. Our task is to determine if the string `s` contains at most one contiguous segment of ones. A contiguous segment of ones is a sequence where the character '1' appears one or more times consecutively without any '0's in between. If such a segment exists and it is the only contiguous segment of ones in the string, we should return `true`. However, if there are multiple segments of ones separated by one or more '0's, we must return `false`.

To provide a couple of examples:

- Given the string "110", we should return `true` because there is only one contiguous segment of ones.
- If we have "1001", we should return `false`, since there are two segments of ones, separated by '0's.

Intuition

The solution to this problem relies on recognizing a pattern within the binary string `s`. Since we need to ensure there is a maximum of one contiguous segment of ones, we are essentially checking if there is ever a transition from '0's back to '1's after the initial segment of ones has ended. This transition will always be indicated by the substring "01" within `s`, which means that there is at least one '0' followed by at least one '1', implying at least two distinct segments of ones.

To come up with the solution, we observe that:

- If the string `s` contains the substring "01", it indicates that a segment of ones has been followed by zeros and later followed by another segment of ones.
- If the substring "01" does not exist in the string `s`, then `s` consists of either all zeros except for an initial segment of ones or contains only ones, fulfilling the requirement of having at most one contiguous segment of ones.

Therefore, the approach is to check for the presence of the substring "01" in `s`. If "01" is not found, then the function returns `true`, indicating that the condition is met. If "01" is found, the function returns `false`, indicating that there are multiple segments of ones.

Here's a quick breakdown:

- If `s` is '11111' or '100000', we return `true` as there are no '01' patterns.
- If `s` is '101' or '110011', we return `false` because we have the '01' pattern indicating multiple segments.

Using these observations, the provided solution:

```
class Solution:
    def checkOnesSegment(self, s: str) -> bool:
        return '01' not in s
```

works perfectly by simply checking for the absence of the pattern "01" in the string `s`.

Solution Approach

The implementation of the solution is quite straightforward and does not require the use of complex algorithms or data structures. The simplicity of the solution stems from the direct relation between the problem's requirement and the pattern that needs to be checked.

Here's an in-depth look at the solution step by step:

- First, we define a class `Solution` which contains the method `checkOnesSegment`. This method is designed to take a string `s` as an input and return a boolean value.
- Inside the method, the only operation is to return the result of the expression `'01' not in s`. This is a direct use of Python's string containment operation which evaluates to `True` if the substring is not found in the larger string, and `False` otherwise.

In other words, the solution uses the built-in string operation to check if the substring `'01'` does not exist anywhere in `s`.

The code is compact because:

- There is no need for loops or recursion, as the existence of a sub-pattern within a string can be determined with the `in` operator in Python, making the process very efficient.
- No additional data structures are needed because we're not manipulating the input string; we're only checking for the presence or absence of a specific pattern.
- There is no need for any complex pattern matching algorithms such as KMP (Knuth-Morris-Pratt), because Python's in-built operations already provide an optimized way to search for substrings.

This approach has O(n) time complexity, where n is the length of the string, since checking for a substring's presence is a linear operation in the size of the string. The space complexity is O(1), as there are no extra space allocations that grow with the input size. The absence or presence of the '01' pattern provides us with a definitive check for our problem criterion.

Therefore, the reference solution approach succinctly checks for the specific pattern that would violate our condition for there to be at most one contiguous segment of ones.

```
class Solution:
    def checkOnesSegment(self, s: str) -> bool:
        return '01' not in s
```

As there is no other content in the Reference Solution Approach section above, this encapsulates the entire solution.

Example Walkthrough

Let's consider the binary string `s = "1000110"`. Here, we want to determine if there is at most one contiguous segment of ones.

Following the provided solution approach, here are the steps we would take to solve the problem:

1. We need to examine the string to find if there is any occurrence of the pattern "01". This pattern is critical because it indicates that a segment of ones is followed by at least one zero and then by another one, constituting multiple segments of ones.
2. We check the string `s`:
   - We see that `s` starts with "1", which is the beginning of a segment of ones.
   - The segment of ones is followed by "0", which could be fine as long as we do not encounter another segment of ones.
   - However, as we continue checking the string, after some zeros, we again encounter "1", and this new segment of ones is preceded by "0", creating the pattern "01".
3. Upon identifying the "01" pattern, we can immediately conclude that there is more than one segment of ones in the string `s`. This is because there is one segment "1" at the start, followed by zeros, and then another segment "11" is observed in the middle of the string.
4. We apply the solution method `checkOnesSegment(s)`:

   ```
   class Solution:
       def checkOnesSegment(self, s: str) -> bool:
           return '01' not in s
   ```

   - In this case, `checkOnesSegment("1000110")` will check for the presence of "01" in the string.
   - The '01' substring is found, so the method will return `False`.

In conclusion, for the example `s = "1000110"`, the evaluation of our method `checkOnesSegment` would yield a `False` result because there are multiple contiguous segments of ones, indicated by the presence of the substring '01' in `s`.

Python Solution

```
class Solution:
    def check_ones_segment(self, s: str) -> bool:
        # Check if the binary string `s` has only one segment of continuous ones.
        # If '01' is not present in the string, it means there is only a single segment of ones.
        return '01' not in s

```

Java Solution

```
class Solution {
    // Method to check if the binary string s has all 1s in a single segment
    public boolean checkOnesSegment(String s) {
        // A binary string has all 1s in a single segment if it doesn't contain "01"
        // Since "01" would indicate that a segment of 1s was ended and followed by 0s,
        // which would mean there is more than one segment of 1s if there are any 1s that follow.
        return !s.contains("01");
    }
}

```

C++ Solution

```
class Solution {
public:
    // Function to check if the binary string has at most one segment of consecutive 1's.
    bool checkOnesSegment(string s) {
        // Check if the string contains the substring "01" which would indicate
        // that there was a transition from 1 to 0, suggesting multiple segments of 1's.
        // If "01" is not found, it means there's only one segment of continuous 1's.
        // In C++, string::npos is the value returned by find when the pattern is not found.
        return s.find("01") == string::npos;
    }
};

```

Typescript Solution

```
/**
 * This function checks whether there is only one segment of consecutive 1's in the binary string without 0's interrupting that segme
 * @param {string} s - The binary string to be checked. It should contain only characters '0' and '1'.
 * @returns {boolean} - Returns `true` if the string contains a single segment of 1's, otherwise returns `false`.
 */
function checkOnesSegment(s: string): boolean {
    // The method includes checks if the string '01' is present in the binary string 's'.
    // If '01' is present, it means there is at least one segment of 1's followed by 0's
    // further followed by another segment of 1's (which we want to avoid, thus we negate the result).
    // If '01' is not present, it means that after the first segment of 1's, only 0's follow
    // or no 0's follow, which is a valid single segment of 1's.
    return !s.includes('01');
};


```

Time and Space Complexity

Time Complexity

The time complexity of the `checkOnesSegment` function is `O(n)`, where `n` is the length of the string `s`. This is because the operation `return '01' not in s` involves a single scan through the string to check for the substring `'01'`. Each character in the string is visited at most once during this process.

Space Complexity

The space complexity of the `checkOnesSegment` function is `O(1)`. This function does not use any additional space that grows with the input size. There are no additional data structures being utilized that depend on the length of `s`. The space used is constant regardless of the input size.