717. 1-bit and 2-bit Characters

Easy Array

Problem Description

In the given problem, we have an array called `bits`, which consists of only the numbers 0 and 1 representing bits. These bits form characters according to certain rules. A single bit of 0 represents one character (one-bit character), and a sequence of two bits that are either 10 or 11 represents another character (two-bit character). The array ends with a 0, which might be a part of the last two-bit character or stand alone as a one-bit character. Our task is to determine if this final 0 is definitely the one-bit character, not part of a two-bit character, by returning `true` if it is, or `false` if it's not.

Intuition

To solve this problem, we can iterate through the `bits` array and track what kind of character we're currently on. Since only a 1 can indicate the beginning of a two-bit character, each time we encounter a 1, we know the next bit belongs to the same character. Therefore, we can increment our position in the array by 2 in this case. If it's a 0, it must represent a one-bit character, so we only increment our position by 1.

The trick of the solution is to notice that we'll never need to skip over the final 0 bit, because it's given that the array ends with a 0. We iterate through the array until we reach the penultimate bit. If our final position falls exactly on the penultimate bit, it means that the last bit is a one-bit character as it's impossible for it to be consumed as part of a two-bit character.

Solution Approach

The implementation of the solution makes use of a simple while-loop to traverse the `bits` array, ending the loop one element before the last to ensure we are only looking at complete characters.

Here's a breakdown of the algorithm, using the given solution as a reference:

1. Initialize an index variable `i` with value 0, which represents the current position in the `bits` array.
2. The length of the array is stored in the variable `n`.
3. We begin a while-loop that continues as long as `i` is less than `n - 1`. We stop at `n - 1` because we're examining characters, and the last character cannot start at the second-to-last bit.
4. Within the loop, we check the value of the current bit:
   - If `bits[i]` is 1, we're at the start of a two-bit character, so the current bit and the next bit form a character. We increment `i` by 2 to jump past this character.
   - If `bits[i]` is 0, we're at a one-bit character, so we increment `i` by 1 to move to the next bit.
5. The loop continues, incrementing `i` in steps of 1 or 2 until it reaches the final or second-to-final bit in the array.
6. After the loop, we check if `i` is exactly at `n - 1`, which is where the final 0 is located.
   - If `i == n - 1`, it means the last bit has not been part of any character we have encountered, so it stands alone as a one-bit character. In this case, we return `true`.
   - If `i != n - 1`, it implies we've gone past the last bit as part of a two-bit character, so we return `false`. However, this situation won't happen as per the problem constraints.

The strength of this solution lies in its simplicity and efficiency - it has a time complexity of O(n), since it involves a single pass through the array, and a space complexity of O(1), as it requires a constant amount of extra space.

Example Walkthrough

Let's consider an example where the `bits` array is `[1, 0, 0]`.

1. We initialize an index variable `i` with the value of 0 and store the length of the array `n` which is 3.
2. Start the while-loop since `i` (0) is less than `n - 1` (2).
3. Check the value of `bits[i]`:
   - `bits[0]` is `1`, which indicates the start of a two-bit character. We increment `i` by 2. Now `i` equals 2.
4. The loop continues since `i` (2) is still less than `n - 1` (2). However, this is not true—it's actually equal, so the condition to continue the loop is not met.
5. Loop has terminated and we check if `i` equals `n - 1` (`i` is 2 and `n - 1` is also 2), which is true.
6. Since `i == n - 1`, we determine that the final 0 is a one-bit character and return `true`.

Now let's consider a slightly variation where `bits` is `[1, 1, 0]`:

1. We initialize `i` with the value of 0 and `n` is again 3.
2. Start the while-loop with `i` less than `n - 1`.
3. Check the value of `bits[i]`:
   - `bits[0]` is `1`, so we have encountered a two-bit character. We increment `i` by 2, resulting in `i` being 2.
4. Since `i` now equals `n - 1` (both are 2), the while-loop condition is no longer met and the loop terminates.
5. We check the condition outside of the loop, where `i == n - 1` holds true.
6. Because the condition `i == n - 1` is true, we conclude that the last `0` represents a one-bit character, so we return `true`.

This illustrates how the solution approach effectively determines whether the final `0` in the `bits` array stands as a one-bit character by iterating through the bits and tracking the position according to the rules described.

Solution Implementation

Python

```python
class Solution:
    def isOneBitCharacter(self, bits: List[int]) -> bool:
        # Initialize the index to 0
        index = 0
        # Get the total number of bits
        total_bits = len(bits)

        # Iterate through the bits until the second-to-last element
        while index < total_bits - 1:
            # Move the index by 2 if the current bit is 1 (since it's the start of a 2-bit character),
            # otherwise by 1 if the current bit is 0 (1-bit character)
            index += bits[index] + 1

        # Return True if the last bit was reached as a 1-bit character,
        # False if we went past it (which is impossible in the constraints given)
        return index == total_bits - 1
```

Java

```java
class Solution {
    public boolean isOneBitCharacter(int[] bits) {
        // Initialize current position index
        int currentIndex = 0;
        // Get the length of bits array
        int length = bits.length;

        // Iterate over the array until we reach the penultimate character
        // since we're checking for a one bit character at the end
        while (currentIndex < length - 1) {
            // If we encounter a '1', we move two places ahead, as '1' signifies the beginning of a two-bit character.
            // If we encounter a '0', we move one place ahead, as '0' signifies a one-bit character.
            currentIndex += bits[currentIndex] + 1;
        }

        // If we are standing at the penultimate position after the loop, it means the last character is a one-bit character
        return currentIndex == length - 1;
    }
}
```

C++

```cpp
#include <vector> // Include the necessary header for std::vector

class Solution {
public:
    // Function to determine if the last character is encoded by one bit
    bool isOneBitCharacter(std::vector<int>& bits) {
        // Initialize the index variable to traverse the array
        int index = 0;
        // Store the size of the bits vector to avoid calculating it multiple times
        int size = bits.size();

        // Loop through the vector until the second-to-last element
        while (index < size - 1) {
            // We increment the index by 2 if the current bit is 1,
            // since it indicates the beginning of a two-bit character,
            // and by 1 if the current bit is 0 (one-bit character).
            index += bits[index] + 1;
        }

        // Return true if we reached exactly the last index (size - 1),
        // which means the last character could only be a one-bit character
        return index == size - 1;
    }
};
```

TypeScript

```typescript
/**
 * Determines if the last character of the bit string is a one-bit character.
 * One-bit character is represented by a single 0, while a two-bit character is represented by 10 or 11.
 * This function assumes that the given string is a valid sequence of one-bit and two-bit characters.
 *
 * @param {number[]} bits - An array of bits.
 * @return {boolean} True if the last character is a one-bit character, false otherwise.
 */
function isOneBitCharacter(bits: number[]): boolean {
    let currentIndex: number = 0;           // Index of the current bit in the loop.
    const length: number = bits.length;     // Total number of bits in the array.

    // Loop through the bits array until the second-to-last element.
    while (currentIndex < length - 1) {
        // If the current bit is 0, it represents a one-bit character, move to the next bit.
        // If it is 1, it represents the first bit of a two-bit character, so skip the next bit.
        currentIndex += bits[currentIndex] + 1;
    }

    // If currentIndex exactly matches the index of the last bit (which would be a one-bit character),
    // it means the last character is a one-bit character.
    return currentIndex === length - 1;
}

// Example usage:
// const result: boolean = isOneBitCharacter([1, 0, 0]);
// console.log(result); // Should log true or false depending on the input
```

```python
class Solution:
    def isOneBitCharacter(self, bits: List[int]) -> bool:
        # Initialize the index to 0
        index = 0
        # Get the total number of bits
        total_bits = len(bits)

        # Iterate through the bits until the second-to-last element
        while index < total_bits - 1:
            # Move the index by 2 if the current bit is 1 (since it's the start of a 2-bit character),
            # otherwise by 1 if the current bit is 0 (1-bit character)
            index += bits[index] + 1

        # Return True if the last bit was reached as a 1-bit character,
        # False if we went past it (which is impossible in the constraints given)
        return index == total_bits - 1
```

Time and Space Complexity

The time complexity of the provided function `isOneBitCharacter` is `O(n)`, where `n` is the length of the input list `bits`. This is because the function uses a while loop that iterates through the bits, incrementing the index `i` by either 1 or 2 each time based on the value of the current bit. In the worst-case scenario, each bit is a 0, and the loop iterates `n` times. In the best case, with alternating 1s and 0s, the function could potentially iterate `n/2` times. However, since we only consider the upper bound, the loop is linear in nature with respect to the input size.

The space complexity of the function is `O(1)`. This is because there are a fixed number of variables (`i`, `n`) that do not depend on the size of the input, and no additional data structures are used that would increase the amount of memory used as the input size increases.