393. UTF-8 Validation

Medium Bit Manipulation Array Leetcode Link

Problem Description

The challenge is to verify if an array of integers, `data`, consisting of the least significant 8 bits to represent a single byte, is a valid UTF-8 encoding sequence. UTF-8 encoding allows characters of lengths varying from 1 to 4 bytes, following specific patterns:

- A 1-byte character starts with a `0`, followed by the actual code for the character.
- An `n-bytes` character sequence begins with `n` ones and a `0`, followed by `n-1` continuation bytes that each starts with `10`.

The patterns for different byte lengths are as follows:

1. 1 byte: `0xxxxxxx`
2. 2 bytes: `110xxxxx 10xxxxxx`
3. 3 bytes: `1110xxxx 10xxxxxx 10xxxxxx`
4. 4 bytes: `11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`

This task is to assess each integer's binary representation in the input array to determine if it correctly represents a UTF-8 character as per the rules above.

Intuition

To resolve whether the data sequence is a valid UTF-8 encoding, one needs to examine each integer and identify the number of bytes the current character should have. This is done by checking the most significant bits (MSB) of each integer:

- If an integer starts with `0`, it should be a single byte character.
- Otherwise, count the number of consecutive `1` bits at the start to know how many bytes the character should have. The count must be at least `2`, and no more than `4`, as UTF-8 can't have more than 4 byte characters.

If it's determined that a character is more than one byte long, the following integers should adhere to the `10xxxxxx` pattern (where the MSB are `10`), indicating that they are continuation bytes for the current character.

The solution iterates over each integer in the data array:

- If we're expecting continuation bytes (`n > 0`), check if the next byte follows the `10xxxxxx` pattern. If not, return `False`.
- If it's the start of a sequence, based on the MSB pattern, determine the total number of bytes the character should consist of and expect that many continuation bytes to follow.
- At any point, if any byte does not meet the expected pattern, return `False`.

After checking all integers, we should not be expecting any more continuation bytes (`n == 0`). If we're still expecting bytes, then a complete character was not formed, and we also return `False`.

This assessment is repeated for each integer in the `data` array to confirm the entire sequence is valid UTF-8.

Solution Approach

The implementation of the solution revolves around bitwise operations, specifically right shifts `>>` and bitwise comparisons. There are no additional data structures required, as we can use scalar variables to track the state as we iterate through the `data` array.

Here's a step-by-step explanation of the algorithm used in the provided solution:

1. Initialize a counter `n` to zero. This counter will track the number of continuation bytes we expect to see for a character encoding.

2. Iterate through each integer `v` in the `data` list.

3. For each integer:

    - If we're expecting continuation bytes (`n > 0`):
        - Check if the integer is a continuation byte by right shifting 6 (`v >> 6`) and comparing if the result equals `0b10` (binary for `10xxxxxx`). If it does not match, return `False`.
        - Decrement `n` by 1, because we have successfully found one of the expected continuation bytes.
    - If we're not expecting a continuation byte (`n == 0`), determine the number of bytes the UTF-8 character should have by checking the patterns:
        - If the integer starts with `0` (`v >> 7 == 0`), it's a 1-byte character, so we continue to the next integer.
        - If the integer starts with `110` (`v >> 5 == 0b110`), it's a 2-byte character. Set `n` to `1` since we expect one continuation byte.
        - If the integer starts with `1110` (`v >> 4 == 0b1110`), it's a 3-byte character. Set `n` to `2` since we expect two continuation bytes.
        - If the integer starts with `11110` (`v >> 3 == 0b11110`), it's a 4-byte character. Set `n` to `3` since we expect three continuation bytes.
        - If none of the above conditions are met, return `False` because the byte does not match any valid UTF-8 starting byte pattern.
4. After processing all integers, we check if `n` equals `0`. If `n` is not `0`, this implies that we were still expecting continuation bytes, and thus the sequence does not represent a properly terminated UTF-8 encoding, so we return `False`. If `n` is `0`, every character has been processed correctly, and we return `True`.

The essence of the solution lies in the careful use of bitwise operations to examine the structure of each byte within an integer and validate that it conforms to the UTF-8 encoding rules.

Example Walkthrough

Let's go through a small example using the provided solution approach. Consider the array `data` with four integers, representing a possible UTF-8 encoded string:

```
data = [197, 130, 1]
```

In binary, these integers are:

```
197 -> 11000101
130 -> 10000010
1   -> 00000001
```

Now, let's apply the solution approach to this example:

1. Initialize `n` to 0. This means we are not expecting any continuation bytes at the beginning.

2. Iterate through each integer in `data`:

3. Take the first integer `197` (`11000101` in binary):

    - `n == 0`, so we're not expecting a continuation byte.
    - Right shift by 5 (`197 >> 5`), the result is `6` in decimal (`110` in binary), matches the `110xxxxx` pattern. This is the start of a 2-byte character, so we set `n = 1` (expecting one continuation byte).
4. Move to the second integer `130` (`10000010` in binary):

    - `n > 0`, so we are expecting a continuation byte.
    - Right shift by 6 (`130 >> 6`), the result is `2` in decimal (`10` in binary), which matches the `10xxxxxx` pattern for a continuation byte. This is correct. We decrement `n` by 1, making `n = 0`.
5. Move to the third integer `1` (`00000001` in binary):

    - `n == 0`, so we are not expecting a continuation byte.
    - Right shift by 7 (`1 >> 7`), the result is `0` in decimal (`0` in binary), which matches the pattern for a 1-byte character (`0xxxxxxx`).
6. Having processed all integers, we check if `n` equals `0` (which it does), indicating that we are not expecting any more continuation bytes and that the sequence represents a properly terminated UTF-8 encoding.

Therefore, according to the solution approach, the given `data` array `[197, 130, 1]` represents a valid UTF-8 encoded string.

Python Solution

```python
class Solution:
    def valid_utf8(self, data: List[int]) -> bool:
        # number of bytes to process
        num_of_bytes = 0

        # Loop through each integer in the data list
        for value in data:
            # If we're in the middle of parsing a valid UTF-8 character
            if num_of_bytes > 0:
                # Check if the first 2 bits are 10, which is a continuation byte
                if value >> 6 != 0b10:
                    # Not a continuation byte, so the sequence is invalid
                    return False
                # We've processed one of the continuation bytes
                num_of_bytes -= 1
            # If we're at the start of a UTF-8 character
            else:
                # Check the first bit; if it's 0, we have a 1-byte character
                if value >> 7 == 0:
                    # Set to 0 as it's a single-byte character
                    num_of_bytes = 0
                # Check the first 3 bits; if they're 110, it's a 2-byte character
                elif value >> 5 == 0b110:
                    # Expecting one more byte for this character
                    num_of_bytes = 1
                # Check the first 4 bits; if they're 1110, it's a 3-byte character
                elif value >> 4 == 0b1110:
                    # Expecting two more bytes for this character
                    num_of_bytes = 2
                # Check the first 5 bits; if they're 11110, it's a 4-byte character
                elif value >> 3 == 0b11110:
                    # Expecting three more bytes for this character
                    num_of_bytes = 3
                else:
                    # The first bits do not match any valid UTF-8 character start
                    return False

        # Check if all characters have been fully processed
        return num_of_bytes == 0

```

Java Solution

```java
class Solution {

    // Function to check if the input data array represents a valid UTF-8 encoding
    public boolean validUtf8(int[] data) {
        int bytesToProcess = 0; // Variable to store the number of bytes to process in UTF-8 character

        // Iterate over each integer in the input array
        for (int value : data) {
            // Check if we are in the middle of processing a multi-byte character
            if (bytesToProcess > 0) {
                // Check if the current byte is a continuation byte (10xxxxxx)
                if ((value >> 6) != 0b10) {
                    return false; // Not a continuation byte, thus invalid
                }
                bytesToProcess--; // Decrement the bytes counter as one more byte has been processed
            } else {
                // Handling the start of a new character
                // Single-byte character (0xxxxxxx)
                if ((value >> 7) == 0) {
                    bytesToProcess = 0; // No bytes left to process, it's a single-byte character
                // Two-byte character (110xxxxx)
                } else if ((value >> 5) == 0b110) {
                    bytesToProcess = 1; // Two-byte character, one more byte to process
                // Three-byte character (1110xxxx)
                } else if ((value >> 4) == 0b1110) {
                    bytesToProcess = 2; // Three-byte character, two more bytes to process
                // Four-byte character (11110xxx)
                } else if ((value >> 3) == 0b11110) {
                    bytesToProcess = 3; // Four-byte character, three more bytes to process
                } else {
                    // If none of the above conditions are met, then it is an invalid leading byte
                    return false;
                }
            }
        }

        // Check if we have processed all the bytes correctly
        return bytesToProcess == 0;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    bool validUtf8(vector<int>& data) {
        int bytesToProcess = 0; // This will keep track of the number of bytes in a UTF-8 character we still need to process.

        // Iterate through each integer in the data array.
        for (int currentValue : data) {
            if (bytesToProcess > 0) {
                // If we are in the middle of processing a multi-byte UTF-8 character, check if the current byte is a continuation
                // A continuation byte starts with the bits 10xxxxxx (binary).
                if ((currentValue >> 6) != 0b10) {
                    return false; // If not, the UTF-8 sequence is invalid.
                }
                --bytesToProcess; // Decrement the counter of bytes left to process.
            } else {
                // If we are not currently processing a UTF-8 character, determine how many bytes the current UTF-8 character consi
                // based on the first byte's most significant bits.

                if ((currentValue >> 7) == 0b0) {
                    bytesToProcess = 0; // If the most significant bit is 0, it's a single-byte character (0xxxxxxx).
                } else if ((currentValue >> 5) == 0b110) {
                    bytesToProcess = 1; // If the first 3 bits are 110, it's a two-byte character (110xxxxx 10xxxxxx).
                } else if ((currentValue >> 4) == 0b1110) {
                    bytesToProcess = 2; // If the first 4 bits are 1110, it's a three-byte character (1110xxxx 10xxxxxx 10xxxxxx).
                } else if ((currentValue >> 3) == 0b11110) {
                    bytesToProcess = 3; // If the first 5 bits are 11110, it's a four-byte character (11110xxx 10xxxxxx 10xxxxxx 10
                } else {
                    return false; // If the byte does not match any of the valid patterns, the sequence is invalid.
                }
            }
        }
        // Return true if all characters have a valid UTF-8 encoding and there are no incomplete characters at the end of the data.
        return bytesToProcess == 0;
    }
};

```

Typescript Solution

```typescript
// Declare a global variable to keep track of the remaining bytes to process in a UTF-8 character.
let bytesToProcess: number = 0;

// Function to validate if a given array of integers represents a valid UTF-8 encoding sequence.
function validUtf8(data: number[]): boolean {
    // Reset the counter for the next validation.
    bytesToProcess = 0;

    // Loop through each integer in the data array to check for UTF-8 validity.
    for (let currentValue of data) {
        if (bytesToProcess > 0) {
            // Check if the current byte is a continuation byte (should start with bits 10xxxxxx).
            if ((currentValue >> 6) !== 0b10) {
                return false; // If not, the sequence is invalid, return false.
            }
            bytesToProcess--; // Decrease the count for bytes left to process.
        } else {
            // Determine the number of bytes in the current UTF-8 character based on its first byte value.
            if ((currentValue >> 7) === 0b0) {
                bytesToProcess = 0; // Single-byte character (0xxxxxxx).
            } else if ((currentValue >> 5) === 0b110) {
                bytesToProcess = 1; // Two-byte character (110xxxxx 10xxxxxx).
            } else if ((currentValue >> 4) === 0b1110) {
                bytesToProcess = 2; // Three-byte character (1110xxxx 10xxxxxx 10xxxxxx).
            } else if ((currentValue >> 3) === 0b11110) {
                bytesToProcess = 3; // Four-byte character (11110xxx 10xxxxxx 10xxxxxx 10xxxxxx).
            } else {
                return false; // If the pattern is not valid for UTF-8, return false.
            }
        }
    }

    // Ensure all characters form a complete UTF-8 encoding sequence with no incomplete characters.
    return bytesToProcess === 0;
}

// Example usage:
// const data = [197, 130, 1];
// console.log(validUtf8(data)); // This should log either 'true' or 'false'.

```

Time and Space Complexity

The given Python code checks if a List of integers represent a valid sequence of UTF-8 encoded characters. It iterates once over all integers (bytes) in the input list to ensure they follow the UTF-8 encoding rules.

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the number of integers in the `data` list. This is because the code iterates over each integer exactly once. Each operation within the loop, such as bit shifting and comparison, is performed in constant time, so the time complexity is linear with respect to the input size.

Space Complexity

The space complexity of the solution is `O(1)`, as the algorithm allocates a constant amount of space: a single counter `n` is used to keep track of the number of bytes that should follow the initial byte in a UTF-8 encoded character. No additional space that scales with the size of the input is used.