1256. Encode Number

Medium Bit Manipulation Math String

Leetcode Link

Problem Description

In this problem, we're given a non-negative integer `num` and we're asked to find its encoded string representation. The encoding process is not given explicitly, but we are expected to infer it from the given table. This table shows the relationship between a sequence of integers and their encoded binary string representation. From this table, we need to figure out the secret function that can convert the input `num` to its encoded form.

Intuition

The intuition behind the solution helps us simplify the problem by recognizing a pattern in binary numbers. When we look at the given table, we can observe that the encoded string for a number appears to be related to the binary representation of that number plus one. More specifically, the encoded string is the binary form of `num + 1` without the first two characters (`'0b'`) which are present in Python to indicate that a number is in binary form, and excluding the leading '1'.

The reason we add one to the number before converting it to binary is that the table starts encoding from zero, but the binary representation of zero is just '0', which after removing the first character would give us an empty string, which is indeed the encoding for zero. This trick allows us to properly align the encoded strings with their binary representation.

Here's the step-by-step approach:

1. Add 1 to the input number to align it with Python's binary representation system.
2. Convert that sum to binary using `bin(num + 1)`.
3. This conversion produces a string starting with '0b', followed by the binary number.
4. We need to truncate the first three characters ('0b1') because the encoded string omits the leading binary digit when it's a '1'.
5. What remains is the encoded representation of the original number.

Using this method, we can encode any non-negative integer as per the problem's requirement.

Solution Approach

The implementation of the solution is quite straightforward and elegant due to the simplicity of the pattern we observed. Here is a breakdown of the implementation steps:

1. **Adding One to `num`:** Since our encoding strategy hinges on the binary representation of `num + 1`, the first step in our code is to increment `num` by one. This aligns the `num` with its respective binary sequence as per the encoding pattern.
2. **Conversion to Binary:**
   - Utilize Python's built-in function `bin()` to convert `(num + 1)` into its binary form, which returns a string.
   - The output of `bin()` for the number `3`, for instance, would be `'0b11'`.
3. **String Slicing:**
   - Since the encoding does not include the '0b' prefix or the leading '1' of the binary representation of `num + 1`, the solution involves slicing the string to remove the first three characters.
   - This is achieved by the expression `[3:]`, which in Python means "give me the string from the third character to the end".
   - For our previous example, slicing `'0b11'` with `[3:]` would result in `'1'`.
4. **Return the Encoded String:**
   - Lastly, the function returns the sliced binary string, which is the encoded representation of the original `num`.

The algorithm does not explicitly use any additional data structures, and its time complexity is O(1) due to direct operations based on the length of the binary representation of `num + 1`. As for space complexity, it remains O(1) as well, since we're only ever dealing with the storage of a single integer and its subsequent string representation.

Here is the Python code implementing this approach:

```python
class Solution:
    def encode(self, num: int) -> str:
        return bin(num + 1)[3:]
```

No additional algorithms or intricate patterns are used in this solution. The encoding relies purely on an understanding of the relationship between the binary representation of an integer and the desired encoded string, and it executes the conversion in just one line of Python code.

Example Walkthrough

Let's take an example to illustrate the solution approach with a specific number. We'll use the number 5 for this purpose.

1. **Adding One to `num`:**
   - First, we add 1 to `num`. Here, `num` is 5, so `num + 1` is 6.
2. **Conversion to Binary:**
   - Next, we use Python's `bin()` function to convert `6` (the result of `num + 1`) into its binary representation.
   - The binary representation of 6 is `'0b110'`.
3. **String Slicing:**
   - According to the encoding rule, we must remove the initial '0b1' from the binary string.
   - Applying the string slicing `[3:]` to `'0b110'` yields `'10'`.
4. **Return the Encoded String:**
   - We then return the resulting string `'10'`, which is the encoded representation of the original number 5.

Now, according to the steps provided in the solution approach, the implementation of the `encode` function would return `'10'` when given the number 5 as input. The method follows the observed pattern of taking the binary representation of `num + 1` and excluding the leading '1' to provide the encoded string.

Python Solution

```python
class Solution:
    def encode(self, number: int) -> str:
        # This function encodes a given number into a binary representation
        # and then truncates the first two characters, '0b', of the binary string.

        # Increment the number by 1 and convert it to a binary string
        binary_string = bin(number + 1)
        # Remove the first two characters '0b' of the binary string
        # and return the modified string
        return binary_string[3:]

```

Java Solution

```java
class Solution {
    // Method to encode a given number to a specific binary string representation
    public String encode(int num) {
        // Convert the number incremented by one to a binary string
        String binaryString = Integer.toBinaryString(num + 1);

        // Remove the first character of the binary string and return the modified string
        // This is because the encoding process requires stripping off the leading '1'
        return binaryString.substring(1);
    }
}

```

C++ Solution

```cpp
#include <bitset>
#include <string>

class Solution {
public:
    // Function to encode an integer to a binary string after incrementing it by 1
    // and removing the leading '1' and all the zeros that precede it.
    string encode(int num) {
        // Increment the input number.
        ++num;

        // Initialize a bitset for 32 bits, representing the incremented number.
        std::bitset<32> bitsetNum(num);

        // Convert the bitset to a binary string.
        std::string binaryString = bitsetNum.to_string();

        // Find the index of the first '1' in the binary representation.
        // This loop skips all the leading '0's.
        size_t firstOneIndex = binaryString.find('1');

        // Extract the substring starting from the character after the first '1',
        // resulting in removing the leading '1' and all zeros before it.
        // This leverages the fact that the binary representation of num + 1 will
        // always have a '1' at the start after incrementing it.
        std::string encodedString = binaryString.substr(firstOneIndex + 1);

        // Return the encoded binary string.
        return encodedString;
    }
};

```

Typescript Solution

```typescript
/**
 * Encode a given number into a binary string with its leading '1' removed.
 * @param num The number to be encoded.
 * @returns A string representation of the binary encoding of num without the leading '1'.
 */
function encode(num: number): string {
    // Increment the number to ensure the binary representation
    // starts with a '1' which we can then remove
    num += 1;

    // Convert the number to a binary string.
    let binaryString: string = num.toString(2);

    // Return the binary string after removing the leading '1'.
    return binaryString.slice(1);
}

```

Time and Space Complexity

The given code consists of a single operation of converting a number to binary and then slicing the string. Here's the complexity analysis:

- **Time Complexity**: The time complexity of the `encode` function is `O(log n)`. The `bin` function, which converts an integer to its binary representation, runs in `O(log n)` time because the length of a binary string representation is proportional to the logarithm of the number (`num`). The slicing operation `[3:]` takes `O(1)` time because slicing operations have a constant time complexity, independent of the slice length.
- **Space Complexity**: The space complexity is also `O(log n)`. This is because the binary string representation of a number takes space that grows with the log of the number. When converting an integer to a binary string, Python creates a new string object which has a length that increases with the size of the input number (`num`). The slicing operation does not significantly add to the space complexity, as it effectively creates a new reference to a substring of the original binary string.