89. Gray Code

Medium Bit Manipulation Math Backtracking

Problem Description

An n-bit Gray code sequence is a special set of binary numbers. This sequence consists of `2^n` integers that satisfy several conditions:

- The integers range from 0 to `2^n - 1`, inclusive, which means that all possible n-bit numbers are included.
- The sequence starts with 0.
- No integer is repeated; each number in the sequence is unique.
- Consecutive numbers in the sequence differ by exactly one bit. This means if you look at the binary representation of any two adjacent numbers, they will be identical except for one bit.
- Finally, the first and the last number in the sequence also differ by exactly one bit. This wraps the sequence around to form a circle, metaphorically speaking, where the end is just one step away from the beginning.

The problem asks us to construct such a sequence for a given `n`.

Intuition

The intuition behind the solution lies in recognising the pattern in which Gray code sequences are generated. A Gray code sequence can be constructed iteratively using a mathematical formula, which is simple yet powerful:

`G(i) = i ^ (i / 2).`

To understand this, observe that for any binary number `i`, when you divide it by 2 (or equivalently shift it right by one bit: `i >> 1`), you are essentially moving all the bits to the right. XOR-ing (`^`) this shifted version with the original number `i` changes exactly one bit, thus satisfying the Gray code property. Also, since the sequence starts from 0, and we do this operation for all numbers from 0 to `2^n - 1`, we ensure no duplicates while also encompassing the entire range.

The provided solution defines a Python function `grayCode` that takes an integer `n` as an input and returns a list of integers representing an n-bit Gray code sequence. The expression `[i ^ (i >> 1) for i in range(1 << n)]` generates the sequence. It uses a list comprehension where each element `i` from 0 to `2^n - 1` (generated by `range(1 << n)`, since `1 << n` is equal to `2^n`) is transformed by the XOR operation with its right-shifted self, yielding the Gray code equivalent for that integer.

Solution Approach

The solution approach leverages the mathematical properties of bitwise operations to efficiently generate the Gray code sequence. It applies the formula `G(i) = i ^ (i >> 1)` directly to ensure the desired properties of the Gray code:

- Only one bit changes at a time.
- The sequence is cyclic, and wraps back to the start after `2^n` elements.

The code uses a `for` loop to iterate from `0` to `2^n - 1`. For each number `i` within this range, the code computes the Gray code number using the formula. The key steps involving algorithms, data structures, or patterns are outlined below:

1. **Bit Shifting**: By shifting `i` to the right by one bit (`i >> 1`), we effectively divide `i` by `2`, achieving a new number whose binary representation is shifted one position.
2. **Bitwise XOR**: The XOR operation (`^`) is then applied to the original number `i` and the right-shifted version of `i`. The property of XOR ensures that the resulting number will differ by exactly one bit from the original number `i`.
3. **List Comprehension**: A list comprehension is used to elegantly apply the Gray code formula to each number in the range and collect the results into a list.
4. **<< Operator**: The `<<` left shift operator is used to calculate `2^n` as `1 << n`, which serves as the upper bound in the `range()` function. This is an efficient way to compute powers of two.

The formula and loop together make the core of the algorithm, relying solely on the fundamental bitwise operations without the need for additional data structures. It's a direct application of the formula with no complex patterns or additional logic necessary, thus the code is both elegant and efficient.

Here is how the loop and Gray code generation works in practice:

```
def grayCode(n: int) -> List[int]:
    # Initializing the result list to store the Gray code sequence.
    # Using list comprehension technique to generate the sequence.
    # For each number i in the range from 0 to (2^n) - 1:
    #   1. Shift the bits of i one position to the right using (i >> 1).
    #   2. Apply XOR operation between i and (i >> 1) to get the Gray code equivalent.
    #   3. Append the resulting Gray code number to the result list.
    return [i ^ (i >> 1) for i in range(1 << n)]
```

By using the properties of bitwise operations, this approach generates a valid Gray code sequence with all the described characteristics in a single, succinct line of Python.

Example Walkthrough

Let's use a small example to illustrate the solution approach by walking through the construction of a 2-bit Gray code sequence.

Given `n = 2`, we aim to generate a sequence that contains `2^n = 2^2 = 4` integers. In binary, these numbers range from `00` to `11`. Here's how we apply the solution:

1. The list comprehension starts with `i = 0`. Computing `G(0)`:
   - i = 0 (in binary `00`)
   - i >> 1 = 0 (in binary `00`)
   - G(0) = i ^ (i >> 1) = `00 ^ 00 = 00` in binary, which is `0` in decimal.
2. Then for `i = 1`, computing `G(1)`:
   - i = 1 (in binary `01`)
   - i >> 1 = 0 (in binary `00`)
   - G(1) = i ^ (i >> 1) = `01 ^ 00 = 01` in binary, which is `1` in decimal.
3. Next for `i = 2`, computing `G(2)`:
   - i = 2 (in binary `10`)
   - i >> 1 = 1 (in binary `01`)
   - G(2) = i ^ (i >> 1) = `10 ^ 01 = 11` in binary, which is `3` in decimal.
4. Finally for `i = 3`, computing `G(3)`:
   - i = 3 (in binary `11`)
   - i >> 1 = 1 (in binary `01`)
   - G(3) = i ^ (i >> 1) = `11 ^ 01 = 10` in binary, which is `2` in decimal.

The sequence, therefore, is `[0, 1, 3, 2]`. Let's verify the properties:

- The integers range from `0` to `2^n - 1` (`0` to `3` for `n = 2`), inclusive.
- The sequence starts with `0`.
- There are no repeated integers.
- Consecutive numbers differ by exactly one bit: Compare `00` with `01`, then `01` with `11`, and `11` with `10`.
- The sequence is cyclical: `0` (`00`) and `2` (`10`) also differ by one bit.

When `n = 2`, this function will generate the sequence `[0, 1, 3, 2]` efficiently by applying the method described in the solution approach. The resulting sequence adheres to all the rules of a Gray code sequence.

Solution Implementation

Python

```
from typing import List

class Solution:
    def grayCode(self, n: int) -> List[int]:
        # Initialize an empty list to store the Gray code sequence
        gray_code_sequence = []

        # Calculate 2^n, the total number of Gray code sequence numbers
        total_numbers = 1 << n

        # Generate Gray code sequence
        for i in range(total_numbers):
            # Apply the formula: binary number XOR (binary number shifted right by one bit)
            # The result is a number in the Gray code sequence
            gray_number = i ^ (i >> 1)

            # Append the Gray code number to the sequence list
            gray_code_sequence.append(gray_number)

        # Return the complete Gray code sequence
        return gray_code_sequence
```

Java

```
class Solution {
    public List<Integer> grayCode(int n) {
        // Initialize an empty list to hold the Gray code sequence
        List<Integer> result = new ArrayList<>();

        // Calculate the total number of elements in the Gray code sequence, which is 2^n
        int totalNumbers = 1 << n;

        // Generate the Gray code sequence
        for (int i = 0; i < totalNumbers; ++i) {
            // The Gray code is generated by XOR-ing the current number
            // with the number right-shifted by one bit
            int grayCodeNumber = i ^ (i >> 1);
            // Add the generated Gray code number to the result list
            result.add(grayCodeNumber);
        }

        // Return the complete list of Gray code sequence
        return result;
    }
}
```

C++

```
#include <vector> // Include the vector header for using the vector container.

class Solution {
public:
    // Function to generate a Gray code sequence for a given number of bits 'n'.
    std::vector<int> grayCode(int n) {
        std::vector<int> results; // Create a vector to store the Gray code sequence.
        int sequenceLength = 1 << n;  // 2^n, the total number of codes in the sequence.

        // Generate the Gray code sequence using the binary to Gray code conversion formula
        // which is G(i) = i ^ (i/2).
        for (int i = 0; i < sequenceLength; ++i) {
            results.push_back(i ^ (i >> 1)); // Apply the Gray code conversion and add it to the result vector.
        }

        // Return the complete Gray code sequence.
        return results;
    }
};
```

TypeScript

```
/**
 * Generates the sequence of Gray codes for a given number of bits.
 * Gray code is a binary numeral system where two successive values differ in only one bit.
 * @param n The number of bits for the Gray code sequence.
 * @return An array that represents the Gray code sequence.
 */
function grayCode(n: number): number[] {
    // Initialize the array that will hold the Gray codes.
    const grayCodes: number[] = [];

    // Calculate 2^n to determine the total number of codes in the sequence.
    const totalCodes: number = 1 << n;

    // Generate the sequence of Gray codes.
    for (let i = 0; i < totalCodes; ++i) {
        // Convert the binary number to a Gray code by performing bitwise XOR
        // of the number with itself right-shifted by one bit.
        grayCodes.push(i ^ (i >> 1));
    }

    // Return the computed Gray code sequence.
    return grayCodes;
}

// Example usage:
// const grayCodeSequence = grayCode(2);
// console.log(grayCodeSequence); // Output: [0, 1, 3, 2]
```

```
from typing import List

class Solution:
    def grayCode(self, n: int) -> List[int]:
        # Initialize an empty list to store the Gray code sequence
        gray_code_sequence = []

        # Calculate 2^n, the total number of Gray code sequence numbers
        total_numbers = 1 << n

        # Generate Gray code sequence
        for i in range(total_numbers):
            # Apply the formula: binary number XOR (binary number shifted right by one bit)
            # The result is a number in the Gray code sequence
            gray_number = i ^ (i >> 1)

            # Append the Gray code number to the sequence list
            gray_code_sequence.append(gray_number)

        # Return the complete Gray code sequence
        return gray_code_sequence
```

Time and Space Complexity

The time complexity of the given code is `O(2^n)` since it is generating all of the Gray codes for an `n`-bit number. Since the number of Gray codes is equal to `2^n`, the loop in the list comprehension will iterate `2^n` times.

The space complexity is also `O(2^n)` because the list that is being returned will contain `2^n` elements, which corresponds to all the possible Gray codes for the `n`-bit number.