1238. Circular Permutation in Binary Representation

Medium · Bit Manipulation · Math · Backtracking

Problem Description

The problem gives us two integers `n` and `start`. It asks us to generate a permutation `p` of the sequence of numbers from 0 to `2^n - 1`, subject to the following conditions:

1. The first element in the permutation `p` must be `start`.
2. Any two consecutive elements in `p` (i.e. `p[i]` and `p[i+1]`) must differ by only one bit in their binary representation. This property is also known as being adjacent in the context of a Gray code sequence.
3. Additionally, the first and last elements of the permutation (`p[0]` and `p[2^n - 1]`) must also differ by only one bit. Essentially, this should be a circular sequence where you can loop from the end back to the beginning seamlessly, maintaining the one-bit difference property.

The task is to return any such valid permutation that satisfies these criteria.

Intuition

To solve this problem, knowledge of the Gray code is quite useful. Gray code is a binary numeral system where two successive numbers differ in only one bit. It's a perfect fit for our problem requirements.

To generate a sequence of Gray code for n bits, we start with a sequence for `n-1` bits then:

1. We prefix the original sequence with `0` to keep the numbers as they are.
2. Then we take the reversed sequence of `n-1` bits and prefix it with `1`, which will flip the most significant bit of those numbers.
3. Finally, we concatenate these two lists into one, which will satisfy the condition that each adjacent number differs by one bit.

However, in this problem, we need to start at a specific number (`start`), and also the sequence should be circular (the start and end elements should differ by only one bit).

We can achieve this by noting that XOR operation between a number and `1` will flip the last bit. Given a number `i`, `i XOR (i >> 1)` will give us the Gray code for `i`. If we further XOR this with `start`, we effectively rotate the Gray code sequence to start at `start` because XOR operation with a number itself cancels out (is 0), while XOR with 0 keeps the number unchanged.

By using the formula `i XOR (i >> 1) XOR start` we can generate a sequence starting from `start` and ensure that the first and last numbers are also one bit apart, satisfying the circular condition:

1. We create a list with a size of `2^n` to hold our permutation.
2. For each number `i` from `0` to `2^n - 1`, we apply the formula to generate the sequence. The `>>` is a right shift bitwise operator, which divides the number by two (or removes the last bit).
3. The resulting list is the desired permutation meeting all problem conditions.

Solution Approach

The implementation of the solution leverages a simple yet clever use of bitwise operations to generate the desired permutation list. The solution does not explicitly construct Gray codes; instead, it uses a known property of Gray codes, which is that the binary representation of the `i`th Gray code is `i XOR (i >> 1)`. Here's a step-by-step of how the algorithm in the reference solution works:

1. The size of the output permutation list will be `2^n`. This is because we want to include all numbers from `0` to `2^n - 1` inclusive.
2. The core of the reference solution relies on list comprehension in Python, which is an elegant and compact way of generating lists.
3. Inside the list comprehension, for every integer `i` in the range `0` to `2^n - 1`, the Gray code equivalent is computed as `i XOR (i >> 1)`. This leverages the bitwise XOR operator `^` and right shift operator `>>`. The right shift operator effectively divides the number by two or in binary terms, shifts all bits to the right, dropping the least significant bit.
4. Having computed the Gray code equivalent, it is further XORed with `start`. This ensures that our permutation will start at the given `start` value. If our Gray code was zero-based, this step essentially "rotates" the Gray code sequence so that the `start` value becomes the first in the sequence. This step is critical because it satisfies the requirement that `p[0]` must be equal to `start`.
5. The list comprehension ultimately constructs the permutation list, with each element now satisfying the property that any two consecutive elements will differ by exactly one bit.

Here's the actual line of Python code responsible for creating the permutation:

```
return [i ^ (i >> 1) ^ start for i in range(1 << n)]
```

In this line of code, `(1 << n)` is equivalent to `2^n`, meaning the `range` function generates all numbers from `0` to `2^n - 1`. The algorithm does not require additional data structures other than the list that it returns, making it space-efficient.

This approach combines knowledge of Gray codes with simple bitwise manipulation in Python to meet all problem requirements efficiently. The resulting algorithm runs in linear time relative to the size of the output list, which is O(2^n), since it must touch each entry in the permutation exactly once.

Example Walkthrough

Let's walk through a small example using the solution approach where `n = 2` and `start = 3`. The sequence we want to generate will have `2^n = 4` elements, and they are permutations of `[0, 1, 2, 3]`. We want `p[0]` to be `3`, and every consecutive element should differ by one bit, including the last element and the first.

Step-by-step process:

1. We calculate the size of the output array, which will be `2^2 = 4`.
2. We know that we must start with `start`, which is `3` in this case, i.e., `p[0] = 3`.
3. Now, we iterate from `i = 0` to `i = 3` and apply the transformation `i XOR (i >> 1) XOR start` to find the rest of the sequence.
4. Let's perform the iterations:
   - For `i = 0`: Gray code is `0 XOR (0 >> 1) = 0`. We then XOR with `start`: `0 XOR 3 = 3`. Our sequence is `[3]`.
   - For `i = 1`: Gray code is `1 XOR (1 >> 1) = 1 XOR 0 = 1`. XOR with `start`: `1 XOR 3 = 2`. The sequence becomes `[3, 2]`.
   - For `i = 2`: Gray code is `2 XOR (2 >> 1) = 2 XOR 1 = 3`. XOR with `start`: `3 XOR 3 = 0`. The sequence updates to `[3, 2, 0]`.
   - For `i = 3`: Gray code is `3 XOR (3 >> 1) = 3 XOR 1 = 2`. XOR with `start`: `2 XOR 3 = 1`. The final sequence is `[3, 2, 0, 1]`.

Each element of this sequence differs by exactly one bit from the next, which you can verify by checking the binary representations: `11 (3)`, `10 (2)`, `00 (0)`, `01 (1)`. Also note that the first and last elements (`3` and `1` respectively) differ by one bit (`11` to `01`), so we have a circular sequence.

Hence, for `n = 2`, `start = 3`, our example has shown that the permutation generated by this approach is `[3, 2, 0, 1]`. This permutation is a valid solution to the problem.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def circularPermutation(self, n: int, start: int) -> List[int]:
        # Create a list of Gray codes with a transformation for circular permutation
        # n: int - The number of digits in the binary representation of the list elements.
        # start: int - The value at which the circular permutation will begin.
        # return: List[int] - The resulting list of circularly permuted Gray codes.

        # Calculate 2^n to determine the total number of elements in the permutation
        total_numbers = 1 << n

        # Generate the list of circularly permuted Gray codes
        circular_perm = [self.grayCode(i) ^ start for i in range(total_numbers)]

        return circular_perm

    def grayCode(self, number: int) -> int:
        # Convert a binary number to its Gray code equivalent
        # number: int - The binary number to convert.
        # return: int - The Gray code of the input number.

        return number ^ (number >> 1)

# Example usage:
# Instantiate the Solution class and call the circularPermutation method
sol = Solution()
# Replace 'n' and 'start' with your specific values
permutation = sol.circularPermutation(n, start)
print(permutation)
```

Java

```java
class Solution {
    // Method to generate and return a list of integers representing a circular permutation in binary representation
    public List<Integer> circularPermutation(int n, int start) {
        // Initialize a list to store the circular permutation result
        List<Integer> answer = new ArrayList<>();

        // Loop to generate all possible binary numbers of n digits
        for (int i = 0; i < (1 << n); ++i) {
            // Generate the i-th Gray code by XORing i with itself right-shifted by 1 bit
            int grayCode = i ^ (i >> 1);
            // XOR the Gray code with the start value to get the circular permutation
            int permutation = grayCode ^ start;
            // Add the permutation to the list
            answer.add(permutation);
        }

        // Return the finished list of permutations
        return answer;
    }
}
```

C++

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

class Solution {
public:
    // Function to generate a circular permutation of size 2^n starting from 'start'.
    std::vector<int> circularPermutation(int n, int start) {
        // Create a vector to hold the numbers of the permutation
        std::vector<int> permutation(1 << n); // 1 << n is equivalent to 2^n

        // Fill the permutation vector using Gray code logic and applying the start offset.
        for (int i = 0; i < (1 << n); ++i) {
            // Calculate the i-th Gray code by XORing i with its right-shifted self.
            int grayCode = i ^ (i >> 1);

            // XOR with 'start' to rotate the permutation so that it begins with 'start'.
            permutation[i] = grayCode ^ start;
        }

        // Return the resulting circular permutation vector.
        return permutation;
    }
};
```

TypeScript

```typescript
// Generates a circular permutation of binary numbers of length n, starting from a given number.
// The approach creates a Gray code sequence and applies bitwise XOR with the start value.
function circularPermutation(n: number, start: number): number[] {
    // Initialize the answer array to hold the circular permutation sequence.
    const permutation: number[] = [];

    // 1 << n computes 2^n, which is the total number of binary numbers possible with n bits.
    // Iterate over the range to generate the sequence.
    for (let index = 0; index < (1 << n); ++index) {
        // Calculate the Gray code for the current index. In Gray code,
        // two successive values differ in only one bit.
        // Then apply the XOR operation with the start value to rotate the sequence
        // such that it begins with 'start'.
        const grayCode = index ^ (index >> 1) ^ start;

        // Append the calculated value to the permutation array.
        permutation.push(grayCode);
    }

    // Return the constructed circular permutation array.
    return permutation;
}
```


```python
from typing import List

class Solution:
    def circularPermutation(self, n: int, start: int) -> List[int]:
        # Create a list of Gray codes with a transformation for circular permutation
        # n: int - The number of digits in the binary representation of the list elements.
        # start: int - The value at which the circular permutation will begin.
        # return: List[int] - The resulting list of circularly permuted Gray codes.

        # Calculate 2^n to determine the total number of elements in the permutation
        total_numbers = 1 << n

        # Generate the list of circularly permuted Gray codes
        circular_perm = [self.grayCode(i) ^ start for i in range(total_numbers)]

        return circular_perm

    def grayCode(self, number: int) -> int:
        # Convert a binary number to its Gray code equivalent
        # number: int - The binary number to convert.
        # return: int - The Gray code of the input number.

        return number ^ (number >> 1)

# Example usage:
# Instantiate the Solution class and call the circularPermutation method
sol = Solution()
# Replace 'n' and 'start' with your specific values
permutation = sol.circularPermutation(n, start)
print(permutation)
```


Time and Space Complexity

Time Complexity

The time complexity of the given code is based on the number of elements generated for the circular permutation. Since the code generates a list of size `2^n` (as indicated by `1 << n` which is equivalent to `2^n`), iterating through all these elements once, the time complexity is `O(2^n)`.

Space Complexity

The space complexity is also `O(2^n)` since a new list of size `2^n` is being created and returned. No additional space that scales with `n` is used, so the space complexity is directly proportional to the output size.