779. K-th Symbol in Grammar

Medium | Bit Manipulation | Recursion | Math

Problem Description

The goal of the problem is to determine the `kth` (1-indexed) symbol in the `nth` row of a special binary table that is constructed according to certain rules. The construction rules are as follows:

- Start with a single digit `0` in the first row.
- For each subsequent row, transform each digit: replace every `0` with `01` and every `1` with `10`.

This pattern of construction creates a sequence of binary strings in each row.

For instance:

- Row 1 would be `0`.
- Row 2 is constructed from Row 1, replacing the single `0` with `01`, making it `01`.
- Row 3 is constructed from Row 2, where `0` becomes `01` and `1` becomes `10`, yielding `0110`.

And so on. The challenge is to determine what the `kth` symbol in the `nth` row is without having to construct the entire rows, which is computationally inefficient for large `n`.

Intuition

The key insight to solve this problem efficiently is recognizing a pattern in the binary sequence of each row, which is related to the nature of binary representations and a property known as the "bit count" or "population count", the number of set bits (1s) in a binary number.

The transformation rules for each row actually describe a recursive formation of binary representation. Each row's sequence can be seen as a binary representation of numbers, where the `nth` row represents the sequence of binary numbers [0, 2^n - 1] with the transformation rules applied.

The critical connection is that the `kth` symbol in the `nth` row corresponds to the parity (even or odd) of the number of 1s in the binary representation of `k-1`.

- If the bit count of `k-1` is even, the `kth` symbol will be `0`.
- If the bit count is odd, the `kth` symbol will be `1`.

Python's `int.bit_count()` returns the number of set bits in the binary representation of a number. By invoking `(k - 1).bit_count()`, we find out how many `1s` are present in the binary representation of `k-1`.

The expression `& 1` is used to determine the parity of the bit count. It performs a bitwise AND with `1`, which just looks at the least significant bit (LSB) of the bit count. Since the LSB will only be `1` for odd numbers, this operation effectively tells us whether there are an even or odd number of `1s`:

- If the LSB is `0`, the bit count is even, and the `kth` symbol is `0`.
- If the LSB is `1`, the bit count is odd, and the `kth` symbol is `1`.

This approach allows us to directly determine the `kth` symbol without explicitly constructing each row, which would be infeasible for large `n`.

The solution implementation uses no additional data structures and relies solely on a single line of code that combines two operations: bit counting and parity checking. Here's a breakdown of the code and the important concepts used:

- **Bit Counting**: The `.bit_count()` method is a Python built-in function that returns the number of `1`s in the binary representation of an integer. It computes the Hamming weight of a number, which is a measure of the number of positions at which the corresponding bits of two binary numbers are different. In the context of this problem, we need to know the bit count of `k - 1` because of the way the binary sequence is constructed in the problem statement. The binary sequence's pattern is directly tied to the bit count of numbers within the sequence.

```
(k - 1).bit_count()  # Returns the number of 1s in the binary representation of k-1.
```

- **Parity Checking**: To find the parity (whether the count is odd or even), we use a bitwise AND operation `& 1`. The result of `(k - 1).bit_count() & 1` will be `1` if the count of `1`s is odd (since the least significant bit would be `1`), and `0` if the count is even (the least significant bit would be `0`). This means that we can directly derive the `kth` symbol in the `nth` row by checking the parity of the bit count of `k - 1`.

```
(k - 1).bit_count() & 1  # Results in 0 if the bit count is even, or 1 if the bit count is odd.
```

As we see in the Reference Solution code below, the function `kthGrammar` simply returns the result of this operation. This line of code captures the entire logic needed to solve the problem. By directly calculating the bit count and its parity, we avoid the construction of the entire table, which reflects the insight gained from the problem's pattern recognition.

```
class Solution:
    def kthGrammar(self, n: int, k: int) -> int:
        return (k - 1).bit_count() & 1
```

In summary, using the innate properties of binary numbers and their arithmetic, this solution exploits the efficient computation tools provided by Python to arrive at the answer with minimal complexity. Such an approach is reminiscent of bitwise operations often used in low-level programming and algorithm optimization.

Let's go through the solution approach with a small example to illustrate how the given algorithm works. Suppose we want to find the `3rd` symbol in the `4th` row of the special binary table. The `4th` row, although not constructed in the solution, would theoretically look like this based on the rules:

1. Row 1: `0`
2. Row 2: `01` (0 → 01)
3. Row 3: `0110` (0 → 01, 1 → 10)
4. Row 4: `01101001` (0 → 01, 1 → 10, 1 → 10, 0 → 01)

So, in the full `4th` row, the sequence is `01101001`, and the `3rd` symbol is `1`.

Now, let's use our solution approach to determine the `3rd` symbol without constructing the entire row.

Step 1: Adjust the index for the `3rd` symbol to work in zero-indexed fashion, which means we consider `k - 1` = `3 - 1` = `2`.

Step 2: Calculate the bit count of `2`. The binary representation of `2` is `10`, which has only `1` set bit.

Step 3: Check the parity of the bit count. Since there is only `1` set bit, the parity is odd.

Step 4: Use the parity to determine the `3rd` symbol:

- An odd bit count implies the symbol is `1`.

Thus, without constructing the entire `4th` row, we quickly determine that the `3rd` symbol in the `4th` row is `1`. This is consistent with the complete sequence we would have theoretically.

Let's verify this with the function provided in the solution:

```
class Solution:
    def kthGrammar(self, n: int, k: int) -> int:
        return (k - 1).bit_count() & 1

# Using the function to find the 3rd symbol in the 4th row
sol = Solution()
print(sol.kthGrammar(4, 3)) # Output: 1, which matches our manual calculation
```

This illustration confirms that our solution approach is both precise and efficient, as we obtained the result without the need for constructing the long binary sequence.

Solution Implementation

```
class Solution:
    def kthGrammar(self, n: int, k: int) -> int:
        # Calculate the bit count (number of set bits) of k-1.
        # In Python, the bit_count method of integers returns the number of 1's in the binary representation of the number.
        bit_count = bin(k - 1).count('1')

        # Determine the k-th element by checking if the bit count is odd
        # The '&' operator is a bitwise AND that results in 1 if the bit_count is odd (since 1 & 1 = 1),
        # and 0 if the bit_count is even (since even_number & 1 = 0).
        return bit_count & 1
```

Java

```
class Solution {
    // Method to find the kth symbol in the nth row of the grammar
    public int kthGrammar(int n, int k) {
        // Calculate the bit count of k-1
        // The bit pattern of k-1 reveals which value will be present at position k
        int bitCountOfKMinusOne = Integer.bitCount(k - 1);

        // If bit count is odd, the kth symbol is 1, otherwise it's 0
        // We use the bitwise AND operator with 1 to get the last bit
        // which will be the answer, 1 for odd and 0 for even
        return bitCountOfKMinusOne & 1;
    }
}
```

C++

```
class Solution {
public:
    // This function returns the k-th symbol in the n-th row of the K-th grammar problem
    int kthGrammar(int n, int k) {
        // The k-th symbol can be determined by counting the number of 1's (set bits)
        // in the binary representation of k - 1, due to the pattern in which the
        // grammar expands. If the count is odd, return 1, otherwise return 0.

        // __builtin_popcount function is used to efficiently count the
        // number of set bits in the integer (k - 1).

        // Then we check if the count of set bits is odd by taking bitwise AND with 1.
        // If the result is 1 (true), it means the count is odd, thus return 1.
        // Otherwise (result is 0), it means the count is even, thus return 0.
        return __builtin_popcount(k - 1) & 1;
    }
};
```

TypeScript

```
// This function returns the k-th symbol in the n-th row of the K-th grammar problem
function kthGrammar(n: number, k: number): number {
    // To find the k-th symbol, we leverage the insight that the sequence can be
    // seen as a binary tree where:
    // - left child is the same as the parent
    // - right child is the inverse of the parent
    // This function uses the number of bits set in (k - 1) to determine the symbol

    // The countSetBits function is defined to count the number of set bits (1s)
    // in the binary representation of a number. This serves the same purpose
    // as __builtin_popcount in the given C++ function.
    function countSetBits(num: number): number {
        let count = 0;
        while (num > 0) {
            // Increment the count for each set bit found
            count += num & 1;
            // Right shift num by 1 to check the next bit
            num = num >> 1;
        }
        return count;
    }

    // Determine the k-th symbol: if the count of set bits in (k - 1) is odd,
    // return 1; if even, return 0. This is done via bit mask and parity check.
    return countSetBits(k - 1) % 2;
}


class Solution:
    def kthGrammar(self, n: int, k: int) -> int:
        # Calculate the bit count (number of set bits) of k-1.
        # In Python, the bit_count method of integers returns the number of 1's in the binary representation of the number.
        bit_count = bin(k - 1).count('1')

        # Determine the k-th element by checking if the bit count is odd
        # The '&' operator is a bitwise AND that results in 1 if the bit_count is odd (since 1 & 1 = 1),
        # and 0 if the bit_count is even (since even_number & 1 = 0).
        return bit_count & 1
```

Time and Space Complexity

The time complexity of the provided code is `O(log k)`, where `log k` is the number of bits needed to represent the integer `k`. This time complexity arises because the `.bit_count()` method counts the number of set bits (1s) in the binary representation of `k - 1`, which takes time proportional to the number of bits in `k - 1`.

The space complexity of the code is `O(1)` because the memory used does not scale with the input size `n` or `k`; only a constant amount of additional space is required for the calculation and storage of the result.