2527. Find Xor-Beauty of Array

Medium Bit Manipulation Array Math

Leetcode Link

Problem Description

This LeetCode problem provides us with an integer array called `nums` which is 0-indexed. It introduces a concept of **effective value** for any triplet of indices in the array `(i, j, k)`. The effective value for these indices is calculated as `((nums[i] | nums[j]) & nums[k])` where `|` represents the bitwise OR operation and `&` represents the bitwise AND operation. The **xor-beauty** of the array is the result of performing a bitwise XOR operation on all the effective values of all possible triplets of indices within the array. The requirement is to calculate and return this xor-beauty value.

Important things to note:

- `0 <= i, j, k < n` indicates that each of the indices must be within the bounds of the array.
- The bitwise OR (`|`) takes two bit patterns and performs the logical inclusive OR operation on each pair of corresponding bits.
- The bitwise AND (`&`) takes two bit patterns and performs the logical AND operation on every pair of corresponding bits.

Intuition

When approaching this problem, one might initially consider iterating through all possible triplets of indices to compute each effective value and then apply the bitwise XOR operation across these values. However, the provided solution suggests a different approach, which implies that there is an observation or property in the bitwise operations that simplifies the problem significantly.

The intuition behind the solution given is that the xor-beauty of the array obtained by computing the effective value for all triplets `(i, j, k)` can be simplified to just perform the bitwise XOR of all elements in the input array `nums`.

This simplified solution suggests that there must be a property of bitwise OR, AND, and XOR that makes every other value except the numbers themselves cancel out when considering all possible combinations of triplets. Essentially, it seems that the problem boils down to xoring each number with itself for all positions `i`, `j`, and `k`, which results in the original value.

Thus, the solution approach is to take advantage of this property and uses Python's `reduce()` function combined with the `xor` operator from the `operator` module to iteratively apply the xor operation on all elements in the array, leading to the xor-beauty directly without explicitly calculating the effective values for all triplets.

Solution Approach

The provided Python code for this problem uses a functional programming style. It employs the use of the `reduce` function from Python's `functools` module. The `reduce` function applies a given function cumulatively to the items of a sequence, from left to right, so as to reduce the sequence to a single value.

The `xor` operator used here is a binary function that performs the "exclusive or" operation, which is imported from Python's built-in `operator` module. This operation takes two bits and returns `1` only if exactly one of the bits is `1`.

Knowing that the `reduce` function will apply the `xor` operator to all elements in the sequence, here is what happens in the given solution:

1. The `reduce` function initializes the reduction with the first element of the `nums` array.
2. It then iteratively applies the `xor` operator to the current result and the next element in the `nums` array.
3. This operation continues for all elements in the `nums` array until it has been reduced to a single value.

The final result after this process is the xor-beauty of the array, which is the cumulative result of xoring all the numbers together.

The use of `reduce` and `xor` indicates no specific need for handling each element position or its relationship with other elements; no iteration over triples or pairs is performed, and no auxiliary data structures are required. This points to an underlying pattern or property within the bitwise operations that allows for this simplified approach.

The code snippet that implements this solution is as follows:

```python
from functools import reduce
from operator import xor

class Solution:
    def xorBeauty(self, nums: List[int]) -> int:
        return reduce(xor, nums)
```

Here, the `xorBeauty` method is defined inside the Solution class and takes the `nums` array as input. It simply returns the xor-beauty, computed by reducing the 'nums' array using bitwise XOR operation.

The key to understanding why this works lies in the properties of bitwise operations, specifically the XOR operation, which include the fact that xoring a number with itself results in zero, and xoring any number with zero results in the number itself. These properties seem to eliminate the need to calculate every single triplet's effective value.

Example Walkthrough

Let's illustrate the solution approach with a small example using an integer array `nums`. Imagine that `nums` is given to us as `[3, 5, 7]`.

Firstly, to follow the problem's statement directly without the optimized solution, we would potentially look at all possible triplets and calculate their effective values, which would be:

- For triplet (i=0, j=0, k=0), the effective value is `((nums[0] | nums[0]) & nums[0])` which equals `(3 | 3) & 3` so `3 & 3` which results in `3`.
- For triplet (i=0, j=0, k=1), the effective value is `((nums[0] | nums[0]) & nums[1])` which equals `3 & 5` resulting in `1`.
- This process would continue for all possible triplets `(i, j, k)` where `i, j, k` range from `0` to `2`.

Considering there are `3` elements, we have `3^3` or `27` such triplet combinations. We would then run the bitwise XOR operation on all these `27` effective values to get the xor-beauty.

But according to our given solution, we don't need to do all that work. Let's apply the simplified solution on our example array `[3, 5, 7]` by performing a cumulative XOR operation on all the elements:

1. We start with the first element, which is `3`.
2. Next, we apply the `xor` operator with the second element: `3 xor 5` which gives us `6`.
3. Lastly, we `xor` this result with the third element: `6 xor 7` which gives us `1`.

So, using the simplified solution, the xor-beauty of the array `[3, 5, 7]` is calculated to be `1`.

The process used in the simplified solution involves no explicit combinatorial effective value calculation and animates the bitwise operations' commutative and associative properties to achieve the same result more efficiently.

Here is how the aforementioned Python code snippet from the solution approach would execute this example:

```python
from functools import reduce
from operator import xor

nums = [3, 5, 7]

xor_beauty = reduce(xor, nums)
print(f"The xor-beauty of the array {nums} is: {xor_beauty}")
```

The output of this code snippet will be: "The xor-beauty of the array [3, 5, 7] is: 1".

Python Solution

```python
from functools import reduce  # Import the 'reduce' function from 'functools' module
from operator import xor      # Import 'xor' function from 'operator' module
from typing import List       # Import 'List' type from 'typing' module for type hinting

class Solution:
    def xor_beauty(self, nums: List[int]) -> int:
        # Perform a pairwise XOR operation on all elements in `nums`
        # The `reduce` function applies the `xor` operation cumulatively to the items of `nums`,
        # from left to right, so as to reduce the iterable `nums` to a single value.
        # Here `xor` is used as the binary operation function.
        return reduce(xor, nums)

# Example usage:
# sol = Solution()
# result = sol.xor_beauty([1, 2, 3, 4])
# print(result)  # This would output the cumulative XOR of the array elements
```

Java Solution

```java
class Solution {
    // Method to calculate the XOR beauty of an array
    public int xorBeauty(int[] nums) {
        // Initialize result variable to hold the cumulative XOR
        int result = 0;

        // Iterate through all elements in the nums array
        for (int number : nums) {
            // XOR the current number with the result so far
            result ^= number;
        }

        // Return the final XOR result which represents the XOR beauty
        return result;
    }
}
```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    // Function to calculate the "xorBeauty" of given numbers.
    int xorBeauty(vector<int>& nums) {
        int result = 0; // Initialize the result to zero.

        // Iterate over each number in the given vector.
        for (const int& number : nums) {
            result ^= number; // Compute the cumulative XOR of the numbers, storing in result.
        }

        // The result is the XOR of all elements in nums.
        return result;
    }
};
```

Typescript Solution

```typescript
// Function that calculates the XOR of all elements in an array.
// The XOR of a single number with itself is 0, and the XOR with 0 is the number itself.
// Hence, the XOR of all numbers in the array gives us the "beauty" of the array.
// @param {number[]} nums - An array of numbers to find the XOR "beauty" of.
// @returns {number} - The result of XOR operation on all the elements of the array.
function xorBeauty(nums: number[]): number {
    // The reduce() method applies a function against an accumulator
    // and each element in the array (from left to right) to reduce it to a single value.
    // Here, it starts with an initial value of 0 and applies XOR operation to all elements.
    return nums.reduce(
        (accumulator: number, current: number) => accumulator ^ current, // XOR operation
        0 // Initial value for the accumulator
    );
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the `nums` list. This is because `reduce` applies the `xor` operation sequentially to the elements of the list, processing each element exactly once.

Space Complexity

The space complexity of the given code is `O(1)`. The `reduce` function uses a constant amount of space to apply the `xor` operation since it only needs to maintain the cumulative xor result and does not require any additional space that depends on the input size.