137. Single Number II

Medium | Bit Manipulation | Array   Leetcode Link

Problem Description

Given an array of integers `nums`, where all elements appear exactly three times except for one unique element that appears only once, the challenge is to identify the unique element. The requirements are to solve this problem with a linear time complexity, meaning the solution should scale proportionally with the size of the input array and use only constant extra space, which implies that the memory used should not scale with input size.

Intuition

To solve this problem efficiently, we need an approach that can process each number and update our tracking variables without saving the entire input or creating additional structures that grow with the input size. Using bitwise operations allows us to operate at the bit level and manipulate individual bits independently of the value or range of the numbers in the array, resulting in a constant extra space usage.

The key understanding in solving this problem is recognizing that if we add up the same bits of all numbers in `nums`, since all but one number appears three times, the sum of bits in any position must be a multiple of three if the unique number does not contribute a bit in that position.

Here's the breakdown of our algorithm using bitwise logic:

- We will use two bitwise markers, `a` and `b`, to keep track of the counts of bits.
- We'll iterate through every bit of each number and update `a` and `b` to keep track of the counts modulo 3.
- The rules for updating `a` and `b` are determined by the current value of `a`, `b`, and the bit value in our current number, `c`.
- We use a series of bitwise AND, OR, and XOR operations to maintain the invariant that after processing each bit of each number, `b` will have a bit set if and only if the corresponding bit in the unique number is set.
- The final answer is the value of `b`, as it represents the bits that are unique to the number appearing only once.

The provided solution successfully uses the digital circuit approach with a simulation of a 3-state machine for each bit position. This approach smartly uses bitwise operators to achieve the linear time complexity with constant extra space, as it does not matter how large the numbers are but rather the number of them.

Solution Approach

The solution approach outlined involves two main algorithms: bit manipulation and the simulation of a digital logic circuit. This combines computer science fundamentals with ideas from electrical engineering to "count" occurrences of bits.

Let's break down the algorithm:

- We initialize two integer variables, `a` and `b`, which are both initially `0`. These variables will represent different states of the count for each bit in our numbers.
- We use a `for` loop to iterate over each number `c` in the given `nums` array. Each iteration involves processing the current number at the bit level.
- For each bit position `i`, we apply bitwise operations to update the state variables `a` and `b`. The rules for updating these variables are based on the current state (`a`, `b`) and the current bit (`c`).

We'll be using this logical circuit to simulate the updates:

- The new value of `a` (`a_i`) is determined by the logical expression `a_i = (~a & b & c) | (a & ~b & ~c)` which corresponds to the truth table conditions for the case when the modulo 3 result is 2.
- The new value of `b` (`b_i`) is determined by `b_i = ~a & (b ^ c)`, which simplifies the updating process.

If we match this against the truth table in the reference, it makes sure that after processing all numbers, `b` contains the bits set only for the unique number that does not appear three times.

The mentioned truth table can be used to extract the logical expressions for new `a_i` and new `b_i`, which are then implemented in the code using bitwise operators AND (`&`), OR (`|`), and XOR (`^`).

It's crucial to realize that the simulation of the logical circuit ensures that with each input (each number in the array), the states `a` and `b` are updated in such a way that all bits that should be discarded are reset after being counted three times.

In the end, `b` will hold the bits that signify the unique number which has not been counted three times, which we return as the result. It's interesting to note that the algorithm is oblivious to the number of bits in the integers involved or the elements' range in the array. It handles the state solely based on the counts per bit, which is why it has a linear runtime and constant space complexity.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Imagine our input array of integers `nums` is `[2, 2, 3, 2]`. Here, all elements appear exactly three times except for `3`, which appears once. We need to find this unique element.

1. We start with two variables `a` and `b`, both set to `0`. These variables track bit counts across the numbers in different states.
2. We process each number in the array, one bit at a time. Since our array is `[2, 2, 3, 2]`, we'll look at these numbers in binary:
   - `2` in binary is `10`
   - `3` in binary is `11`
3. For the first number (which is `2` or `10` in binary):
   - We perform bitwise operations to update `a` and `b`.
   - As per our rules, after processing, `a` remains `0` and `b` becomes `10` (2 in decimal), since we're seeing `2` for the first time.
4. We process the second number (also `2`):
   - We update `a` and `b` again. Now, `a` becomes `10` and `b` resets to `0`, which tracks that we've seen `2` twice.
5. The third number is `3` (`11` in binary):
   - Updating `a` and `b` with `3` changes nothing in `a`, because the unique bits brought by `3` don't align with bits from `2` which were in `a`. For `b`, it absorbs the new unique bits, so `b` changes to `11`.
6. Finally, we process the last number, another `2`:
   - This time, the bits in `a` align with the incoming bits of `2`, and both variables `a` and `b` get updated. `a` will be reset to `0`, and `b` will also be reset to `0` for the bits where `2` contributed, leaving only the bits from the unique number which is `3`.
7. After the final iteration, `a` is `0`, and `b` is `11` (which is `3` in decimal). This final value of `b` is the unique element that does not appear three times in the array.

So, we conclude that the unique element in the array `[2, 2, 3, 2]` is `3`.

This walk-through demonstrates that at each iteration, the algorithm correctly updates the state variables `a` and `b` by applying the rules from our truth table and logical expressions, leading to the correct identification of the unique number with linear time complexity and constant space usage.

Python Solution

```python
class Solution:
    def singleNumber(self, nums: list[int]) -> int:
        # `once` tracks bits that have appeared once
        # `twice` tracks bits that have appeared twice
        once = twice = 0

        for num in nums:
            # Update `once` and `twice` with current number `num`
            # `once` should only hold bits that are exactly seen once so far
            # And `twice` should only hold bits that are exactly seen twice so far
            # Bits seen three times should be cleared from both `once` and `twice`

            # First, calculate `once` considering the current number `num`
            # Use `twice` to mask bits that are already seen two times before
            # because we want to ignore the third time

            # The operations can be explained as:
            # If `once` is already set, and current bit of num is 0, keep `once`
            # If `once` is not set, and `twice` and current bit is set, set `once`
            # This updates bit in `once` only if bit in `num` is 1 and wasn't part
            # of `twice`, or if bit in `num` is 0 and bit was already set in `once`.
            once_new = (~once & twice & num) | (once & ~twice & ~num)

            # Next, calculate `twice` considering the current number `num`
            # Bits in `once` are used to clear bits that appear for the third time
            # This updates bits in `twice` only if bit in `num` is different from bit in `twice`
            # and bit in `once` is 0, to ensure it's not the third time we see this bit.
            twice_new = ~once & (twice ^ num)

            # Update `once` and `twice`
            once, twice = once_new, twice_new

        # Return the number that appears only once
        # All bits in `once` have now appeared either 0 or 3 times, which will end up with 0
        # All bits in `twice` have now appeared exactly once or twice, which end up with 0
        # Only the single number will set bits in `twice`
        return twice

```

Java Solution

```java
class Solution {
    public int singleNumber(int[] nums) {
        int bit1 = 0;  // This will hold the XOR of all the elements which appear exactly once
        int bit2 = 0;  // This will hold the XOR of all the elements which appear exactly twice

        for (int num : nums) {
            // These intermediate values store the new values of bit1 and bit2
            int newBit1 = (~bit1 & bit2 & num) | (bit1 & ~bit2 & ~num);
            int newBit2 = ~bit1 & (bit2 ^ num);

            // Update bit1 and bit2 with their new values for the next iteration
            bit1 = newBit1;
            bit2 = newBit2;
        }

        // At the end, bit2 will be the value that appears exactly once as
        // bit1 will store the XOR of the element which appears thrice which is 0.
        return bit2;
    }
}

```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    int singleNumber(std::vector<int>& nums) {
        int onlyOnce = 0; // Variable for the element that appears only once
        int twiceState = 0; // Variable for the element that appears twice

        // Iterate over each element in the input vector
        for (int num : nums) {
            // Update `onlyOnce` only if `twiceState` is not set. This is part of tracking
            // the element that appears once. If `twiceState` is set, reset `onlyOnce`.
            int newOnlyOnce = (~onlyOnce & twiceState & num) | (onlyOnce & ~twiceState & ~num);

            // Update `twiceState`: it should be set if `onlyOnce` is not set and `num` is unique (XOR operation).
            // If `num` is already in `onlyOnce`, then it should be cleared.
            int newTwiceState = ~onlyOnce & (twiceState ^ num);

            // Update the states with the newly calculated values
            onlyOnce = newOnlyOnce;
            twiceState = newTwiceState;
        }
        // Since the problem guarantees one number appears exactly once and the others appear exactly three times,
        // the `onlyOnce` variable will end up with the single appearing number.
        return twiceState; // The element that appears once is found in `twiceState` at the end of the loop
    }
};

```

Typescript Solution

```typescript
function singleNumber(nums: number[]): number {
    let bitwiseA = 0; // Initialize 'bitwiseA' which will hold a bitwise representation
    let bitwiseB = 0; // Initialize 'bitwiseB' which will hold another bitwise representation

    // Loop through each number in the array
    for (const num of nums) {
        // Calculate the new value of 'bitwiseA' based on current 'bitwiseA', 'bitwiseB', and current num.
        // It uses bitwise NOT (~), AND (&), and OR (|) operations.
        const newBitwiseA = (~bitwiseA & bitwiseB & num) | (bitwiseA & ~bitwiseB & ~num);

        // Calculate the new value of 'bitwiseB' based on current 'bitwiseB' and num.
        // It uses bitwise NOT (~), AND (&), and XOR (^) operations.
        const newBitwiseB = ~bitwiseA & (bitwiseB ^ num);

        // Update 'bitwiseA' and 'bitwiseB' with the newly computed values.
        bitwiseA = newBitwiseA;
        bitwiseB = newBitwiseB;
    }

    // Since every element appears three times except for one element which appears once,
    // 'bitwiseB' will hold the single number that appears only once at the end of the loop.
    return bitwiseB;
}

```

Time and Space Complexity

The time complexity of the provided code is `O(n)`, where `n` is the length of the input list `nums`. This is because there is a single loop that iterates through all the elements of the array once.

The space complexity of the code is `O(1)` as it uses a constant amount of space. The variables `a`, `b`, `aa`, and `bb` do not grow with the input size, hence the space used remains constant regardless of the size of the input array.