136. Single Number

Easy | Bit Manipulation | Array

Problem Description

In this problem, you're given an array of integers `nums` where each element appears exactly twice, except for one element which appears only once. The task is to identify that single element that does not have a pair.

The conditions are:

1. The array is not empty (it contains at least one number).
2. The runtime complexity of your solution should be linear, in other words, O(n), which implies you can only traverse the array once.
3. You must use only constant extra space, so no additional arrays or data structures that grow with the size of the input.

Understanding the problem, the first thing that might come to mind is to count the occurrences of each number and return the one that has a count of one. However, this would require additional space proportional to the number of distinct elements, violating the space complexity requirement.

Therefore, we need a clever approach that uses a single variable to keep track of the unique element while traversing the array once. This is where bit manipulation comes in handy.

Intuition

The solution hinges on the concept of the XOR bitwise operation, which has two key properties that we can exploit:

1. Any number XORed with itself equals 0, i.e., `a XOR a = 0`.
2. Any number XORed with 0 equals the number itself, i.e., `a XOR 0 = a`.

Given these properties, we can iterate over all the numbers in the array and XOR them progressively. The key insight is that pairs of identical numbers will cancel each other out because of property 1 (resulting in 0), and single numbers will emerge unchanged because of property 2.

Imagine if we have an array like `[2, 1, 4, 1, 2]`. If we apply XOR to all elements sequentially, we'll end up with `2 XOR 1 XOR 4 XOR 1 XOR 2`. The order of XOR operations doesn't matter due to its associative property, so we can group the identical numbers: `(2 XOR 2) XOR (1 XOR 1) XOR 4`. Now, using the first property, identical numbers XORed result in 0: `0 XOR 0 XOR 4`. Finally, applying the second property, we are left with `4`, which is the single number we're looking for.

Solution Approach

Referencing the Reference Solution Approach outlined earlier, the implementation uses a single Python function `singleNumber` that receives the list `nums` as its argument. To solve the problem, no explicit data structures are instantiated; instead, we leverage the `reduce` function and the `xor` from `operator` as the means to apply the XOR operation across all elements in the list.

The `reduce` function is a functional programming tool that applies a function of two arguments cumulatively to the items of a sequence, from left to right, so as to reduce the sequence to a single value. For our XOR operation, `reduce` is perfectly suited because it will take each element in `nums` in turn and apply the `xor` function, essentially folding the sequence down to one number through the cumulative XOR process.

Here's a step by step look at how `reduce` works in our context:

1. It initializes the "accumulator" (the running result of the XOR operations) with the first element of `nums`.
2. It then takes the next element and applies the `xor` function to this element and the accumulator.
3. The result of this operation becomes the new value of the accumulator.
4. This process repeats for each element in `nums` until all elements have been combined into one value by the XOR operation.

The `xor` function used here acts as our XOR logic, taking two numbers and returning their bitwise XOR result. In Python, the `xor` operation is represented by the caret symbol `"^"`.

The final implementation looks like this:

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

class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        # Apply reduce to perform XOR on all elements of nums
        # This will leave us with the unique element
        return reduce(xor, nums)
```

By applying the XOR operation across all numbers using `reduce`, we're left with the unique number that doesn't have a duplicate in the list. Since the XOR of a number with itself is zero and the XOR of a number with zero is the number itself, all the paired numbers will cancel out, and only the unique one will remain. This aligns perfectly with the constraints of the problem, leading to a linear time complexity solution with constant space complexity.

Example Walkthrough

Let's take a small example to illustrate the solution approach. Imagine we have an array `nums = [5, 3, 5, 4, 3]` and we want to find the single element that appears only once using the XOR operation.

1. We start with the first element (5) and initialize our accumulator with this value: `accumulator = 5`. Currently, our XOR chain looks like this: `5`.
2. Move to the second element (3) and perform XOR with the accumulator: `accumulator = 5 ^ 3`. The XOR chain now looks like this: `5 ^ 3`.
3. Next, go to the third element (another 5) and perform XOR: `accumulator = (5 ^ 3) ^ 5`. Since XOR of a number with itself is zero, it simplifies to: `3`.
4. Proceed to the fourth element (4) and perform XOR: `accumulator = 3 ^ 4`. The XOR chain is now: `3 ^ 4`.
5. Finally, take the last element (another 3) and perform XOR: `accumulator = (3 ^ 4) ^ 3`. This simplifies to `4`, because XOR of a number with itself (3 in this case) cancels out.

At the end of this process, the accumulator holds the value `4`, which is the number that appears only once in the array. Each step applies XOR to the current accumulator and the next element in the array, effectively canceling out pairs of numbers and leaving the unique number alone.

With the XOR properties in mind, it's clear that all matched pairs in the array will result in 0, and when 0 is XORed with the unique element, it will result in that unique element itself.

The Python implementation using `reduce` applies this process across the entire `nums` array, and we are directly left with the single number as the result, which is `4` in this case. This is the number that this solution approach will correctly identify as the single element.

Python Solution

```python
from functools import reduce  # Import reduce from functools module
from operator import xor  # Import bitwise xor operator

class Solution:
    def singleNumber(self, nums):
        # This method finds the element that appears only once in an array
        # where every other element appears exactly twice.

        # It applies the reduce function to perform a cumulative xor operation
        # over all the numbers in the list. The xor of a number with itself is 0,
        # and the xor of a number with 0 is the number itself. Thus, only the
        # element that appears once will remain after the reduction.

        return reduce(xor, nums)
```

Java Solution

```java
class Solution {
    // Function to find the single number in an array where every element appears twice except for one
    public int singleNumber(int[] nums) {
        // Initialize the variable 'answer' with 0
        int answer = 0;

        // Loop over each value in the array 'nums'
        for (int value : nums) {
            // Apply XOR operation between the 'answer' and the 'value'
            // Since XOR of a number with itself is 0 and with 0 is the number itself,
            // this will cancel out all pairs leaving the single number alone
            answer ^= value;
        }

        // Return the single number that doesn't have a pair
        return answer;
    }
}
```

C++ Solution

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

// Define the 'Solution' class.
class Solution {
public:
    // Function to find the single number in a vector of integers.
    // It utilizes bitwise XOR to find the unique element.
    int singleNumber(std::vector<int>& nums) {
        int singleNumber = 0; // Initialize the variable to store the result.

        // Loop over each element in the vector.
        for (int num : nums) {
            singleNumber ^= num; // Apply XOR operation between the current result and the current number.
        }

        return singleNumber; // Return the result, which is the single number.
    }
};
```

Typescript Solution

```typescript
/**
 * Finds the single number in an array of integers where each element appears twice except for one.
 * @param {number[]} numbers - An array of numbers where all except one number are present twice.
 * @returns {number} - The single number that appears only once in the array.
 */
function singleNumber(numbers: number[]): number {
    // Use the reduce method to iterate through the array, applying the XOR (^) operator.
    // The XOR operator will effectively cancel out duplicates, leaving the unique number.
    // Starting value for the reduce function (second parameter) is 0.
    return numbers.reduce((accumulator, currentValue) => accumulator ^ currentValue, 0);
}

// Usage example:
// const numbersArray = [4, 1, 2, 1, 2];
// const result = singleNumber(numbersArray);
// console.log(result); // Outputs the single number, in this case 4
```

Time and Space Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the array `nums`. This is because the `reduce` function applies the `xor` operation sequentially to the elements of the list, which requires a single pass through all the elements.

The space complexity of the code is `O(1)`. The space used does not depend on the size of the input list, as the `xor` operation is applied in-place and the `reduce` function only needs a constant amount of space to store the intermediate result of the `xor` operations.