645. Set Mismatch

Easy | Bit Manipulation | Array | Hash Table | Sorting

Leetcode Link

Problem Description

In this problem, we're dealing with a particular set of integers ranging from `1` to `n` where due to an error, one of the numbers was duplicated, causing another number to be missing. Our task is to figure out which number was duplicated (appears twice) and which one is missing from the array `nums` given to us.

The key to solving this problem lies in taking advantage of the properties of exclusive or (XOR) operations and the specifics of the data corruption that occurred.

Intuition

The intuition behind the solution can be developed by considering the properties of XOR operation. We know that for any number `x`, doing `x XOR x` gives us `0`, and `x XOR 0` gives us `x`. Thus, XOR-ing a number with itself cancels it out.

With this property in mind, we can XOR all elements in the `nums` array with each integer from `1` to `n`. Ideally, if no number was duplicated and none missing, the result should be `0` because every number would cancel itself out. However, since we have one duplicate and one missing number, we'll be left with the XOR of these two numbers.

From there, we can distinguish between the duplicate and missing numbers by noticing that they must differ in at least one bit. We find this differing bit (the rightmost bit is usually a good choice), and use it as a mask to separate the numbers into two groups - one with this bit set and one without it.

Now, we apply XOR operation within each group, including the numbers from `1` to `n`. This separate XORing would give us the duplicate number and the missing number, but we don't know which is which yet. We verify by checking each number in `nums` again. The number that we come across in the array is the duplicate, and the other one is the missing number.

By applying XOR in such strategic ways, we leverage its ability to cancel equal pairs and isolate the odd ones out: in this case, our duplicate and missing numbers.

Solution Approach

There are multiple approaches to solving this problem, each using different algorithms, data structures, or patterns. Here, we'll discuss three distinct approaches.

Solution 1: Mathematics

Using the principle of arithmetical sums, we can find the difference between the sum of a complete set from `1` to `n` (`s_1`) and the sum of our corrupted array (`s_2` after duplicates removal). The sum difference `s - s_2` gives us the repeated element, while `s_1 - s_2` identifies the missing element. This approach is direct and relies on simple arithmetic operations. However, it requires handling potentially large sums, which could lead to integer overflow issues if not careful.

Solution 2: Hash Table

This approach is more straightforward: construct a hash table to count the number of times each element appears in `nums`. Then, simply iterate through the range `1` to `n` and check the count for each number. If the count is `2`, the number is duplicated; if `0`, it's the missing number. The hash table provides a direct mapping from elements to their counts, making lookups efficient (`O(1)` on average).

Solution 3: Bit Operation (implemented in the provided code)

The bit manipulation solution is more complex but efficient, especially regarding space complexity. The idea revolves around XOR operations. We iterate through the array `nums` and range from `1` to `n`, XOR-ing each element and index. This process will give us `xs`, the XOR of the duplicate number `a` and the missing number `b (xs = a XOR b)`.

To separate these two values, we find the rightmost bit that is set in `xs` (`lb = xs & -xs`). We use this bit to divide the numbers into two groups and perform XOR operations on those groups. After separating elements into two piles, we XOR all elements that have this bit set and all elements that don't. This will result in two numbers, one being `a` and the other `b`. To identify which is which, we iterate through the `nums` array again. The one we find in the array is our duplicate number.

Finally, we return the duplicate and missing numbers as a list [`a`, `b`], ensuring the order is correct based on the final check with the original array.

The brilliance of this approach lies in using the distinct patterns of XOR to segregate and identify unique elements while maintaining a very low space complexity.

Example Walkthrough

Let's illustrate the third solution approach with XOR operation using a small example.

Assume our array `nums` is `[1, 2, 2, 4]` for `n = 4`. The ideal array without any errors would be `[1, 2, 3, 4]`.

1. We start by initializing `xs` to `0` and we XOR all elements in `nums` and all numbers from `1` to `n`. Doing so gives us:

   `xs = 1^2^2^4^1^2^3^4`

   After canceling out duplicate pairs `(1^1, 2^2, and 4^4)` we are left with:

   `xs = 2^3`

   `xs` is now the XOR of the duplicate number and the missing number.

2. Next, we find the rightmost set bit in `xs`, which we'll use as a bitmask to differentiate between the two numbers that didn't cancel out.

   `xs = 2^3 = 0010^0011 = 0001`

   The rightmost set bit is `1` (we can find it by performing `xs & -xs`).

3. We now use this bit as a mask to split the elements into two groups. The first group will have this bit set (`3` for our case) and the second will not (`2` for our case).

   We perform XOR operations separately on these two groups alongside the numbers from `1` to `n`:

   - With bit set: `3`
   - Without bit set: `1^2^2^4^1^4`

   After cancellation:

   - With bit set: `3` (already isolated, no duplicates to cancel it out)
   - Without bit set: `2` (since `1^1` and `4^4` cancel themselves out leaving `2^2` which, after XORing, gives `2`)
4. Now, we have two results from each group: `3` and `2`. We don't yet know which is the duplicate number and which is the missing one.

5. To find out, we have to compare it against the `nums` array. In our example, we find `2` in `nums`, meaning `2` is the duplicate number.

6. Since `3` did not appear in `nums`, it is the missing number.

7. We conclude that the duplicate number (`a`) is `2`, and the missing number (`b`) is `3`. Our ordered result would be `[2, 3]`.

In this example walkthrough, we have effectively demonstrated how the third solution approach, which focuses on bit manipulation, can identify the duplicate and the missing numbers in an array with a corrupted sequence of integers.

Python Solution

```python
class Solution:
    def findErrorNums(self, nums: List[int]) -> List[int]:
        xor_result = 0
        # First pass: calculate the XOR of all indices and the elements in nums
        for index, value in enumerate(nums, 1):
            xor_result ^= index ^ value

        # Obtain the rightmost set bit (least significant bit)
        # This will serve as a mask for segregating numbers
        rightmost_set_bit = xor_result & -xor_result

        # We will use `number_a` to find one of the numbers by segregating into two buckets
        # The rightmost set bit will allow us to xor numbers into two groups
        # One group where the bit is set and another where it's not
        number_a = 0
        for index, value in enumerate(nums, 1):
            if index & rightmost_set_bit:
                number_a ^= index
            if value & rightmost_set_bit:
                number_a ^= value

        # `number_b` is the other number we need to find
        number_b = xor_result ^ number_a

        # Check if `number_a` is the duplicated number or the missing number
        # If `number_a` is found in nums, it is the duplicated one
        for value in nums:
            if value == number_a:
                return [number_a, number_b]

        # If `number_a` is not in nums, it's the missing number and `number_b` is the duplicate
        return [number_b, number_a]

```

Java Solution

```java
class Solution {
    public int[] findErrorNums(int[] nums) {
        // The length of the array
        int length = nums.length;
        // Variable to store the XOR of all indices and values
        int xorResult = 0;

        // XOR all the indices with their corresponding values
        for (int i = 1; i <= length; ++i) {
            xorResult ^= i ^ nums[i - 1];
        }

        // Find the rightmost set bit of xorResult
        int rightmostSetBit = xorResult & -xorResult;

        // Initialize variables to 0 that will store the two unique numbers
        int number1 = 0;

        // Separate the numbers into two groups and XOR them individually
        for (int i = 1; i <= length; ++i) {
            // Check for the rightmost set bit and XOR
            if ((i & rightmostSetBit) > 0) {
                number1 ^= i;
            }
            if ((nums[i - 1] & rightmostSetBit) > 0) {
                number1 ^= nums[i - 1];
            }
        }

        // XORing with xorResult to find the other number
        int number2 = xorResult ^ number1;

        // Iterate over the array to check which one is the duplicate number
        for (int i = 0; i < length; ++i) {
            if (nums[i] == number1) {
                // If number1 is the duplicate, return number1 and number2 in order
                return new int[] {number1, number2};
            }
        }

        // If number1 is not the duplicate then number2 is, so return number2 and number1 in order
        return new int[] {number2, number1};
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    vector<int> findErrorNums(vector<int>& nums) {
        int size = nums.size(); // Get the size of the input vector
        int xorSum = 0; // Initialize variable for XOR sum

        // Calculate XOR for all numbers from 1 to n and all elements in nums
        for (int i = 1; i <= size; ++i) {
            xorSum ^= i ^ nums[i - 1];
        }

        // Determine the rightmost set bit (which differs between the missing and duplicate)
        int rightMostBit = xorSum & -xorSum;
        int oneBitSetNum = 0; // This will hold one of the two numbers (missing or duplicate)

        // Divide the numbers into two groups based on the rightMostBit and XOR within groups
        for (int i = 1; i <= size; ++i) {
            if (i & rightMostBit) { // If the bit is set
                oneBitSetNum ^= i; // XOR for the range 1 to N
            }
            if (nums[i - 1] & rightMostBit) { // If the bit is set in the current element
                oneBitSetNum ^= nums[i - 1]; // XOR with the nums elements
            }
        }

        int otherNum = xorSum ^ oneBitSetNum; // Find the second number using the xorSum

        // Determine which number is the missing number and which one is the duplicate
        for (int i = 0; i < size; ++i) {
            if (nums[i] == oneBitSetNum) {
                // If we find oneBitSetNum in nums array, it is the duplicate
                return {oneBitSetNum, otherNum}; // Return the duplicate and missing numbers
            }
        }

        // If not found, then oneBitSetNum is the missing number and otherNum is the duplicate
        return {otherNum, oneBitSetNum};
    }
};

```

Typescript Solution

```typescript
function findErrorNums(nums: number[]): number[] {
    // Calculate the total number of elements in the array.
    const length = nums.length;

    // Initialize xorSum to 0, which will be used to find the XOR sum of all elements and their indices.
    let xorSum = 0;

    // Loop through the array and XOR both the index and value of each element.
    for (let index = 1; index <= length; ++index) {
        xorSum ^= index ^ nums[index - 1];
    }

    // Determine the rightmost set bit in xorSum using two's complement.
    const rightmostSetBit = xorSum & -xorSum;

    // Initialize a variable to hold one of the two numbers we're looking for.
    let oneNumber = 0;

    // Loop through the array and indices to partition them based on the rightmostSetBit.
    for (let index = 1; index <= length; ++index) {
        if (index & rightmostSetBit) {
            oneNumber ^= index;
        }
        if (nums[index - 1] & rightmostSetBit) {
            oneNumber ^= nums[index - 1];
        }
    }

    // Determine the other number by XORing oneNumber with xorSum.
    const otherNumber = xorSum ^ oneNumber;

    // Check which number is missing from the array and which number is duplicated.
    // The number that appears in the nums array is the duplicated one,
    // and the one that's missing is the number that should have been there.
    return nums.includes(oneNumber) ? [oneNumber, otherNumber] : [otherNumber, oneNumber];
}

```

Time and Space Complexity

The time complexity of the given code is `O(n)` where `n` is the length of the array `nums`. This is because each of the loops in the function (`for i, x in enumerate(nums, 1):` and `for x in nums:`) iterates over the list `nums` exactly once, and all operations within the loops are constant time operations.

The space complexity of the function is `O(1)`, which means it uses a constant amount of extra space. Despite the input list size, the space consumed by the variables `xs`, `lb`, `a`, and `b` does not scale with `n`, thus having no additional space depending on the input size.