268. Missing Number

Easy | Bit Manipulation | Array | Hash Table | Math | Binary Search | Sorting

Leetcode Link

Problem Description

The task is to find a single missing number in an array that contains `n` unique numbers from the range `[0, n]`. This means the array is one number short of the full set, and your job is to identify which number from the range is not present in the array. For example, if the array `nums` is given as `[3,0,1]`, the number `2` is not included in the array but it is in the range `[0, 3]`, so the output should be `2`.

Intuition

The provided solution uses a bitwise XOR operation to find the missing number. XOR is a logical operation that outputs true or `1` only when the number of true inputs is odd. For two bits, it outputs `1` if the bits are different, and `0` if they are the same. When you XOR the same number together, the result is `0`. If you XOR a number with `0`, you get the number back. We use this property to our advantage.

The solution uses an XOR operation between matching indices and values from the range `[0, n]` and the elements in the array `nums`. Since the array is guaranteed to have `n` unique numbers and we're doing XOR with `n+1` numbers (since the range is `[0, n]`), the number that is missing will be the one that does not have a pair and thus will be left after all XOR operations are done.

The solution code first enumerates over `nums` starting with `1` instead of `0` since the range in the problem is `[0, n]` and you want to XOR each number with its index. The operation `^=` is used to apply XOR in place. The `reduce` function from the `functools` module then applies the XOR operation cumulatively to the items of the iterable — which in this case is each XOR operation of the index and value — and thus cumulatively XORs all the indices and values, resulting in the missing number.

Solution Approach

The solution for finding the missing number from the array `nums` involves a clever use of the `XOR` bitwise operation. As explained in the intuition section, the properties of XOR can be wielded to pinpoint the missing integer. The `reduce` function and the `xor` function from the `functools` module in Python are used to apply the XOR operation across an iterable, which is a generator expression in this case.

Here's a step-by-step breakdown of the implementation:

1. The `enumerate` function is used to generate pairs of index and value from the array `nums`. The index starts from `1` instead of `0` to align with our XOR operation approach which requires an extra element from the range `[0, n]`.
2. The generator expression `(i ^ v for i, v in enumerate(nums, 1))` computes the XOR of the index and value for each element in the `nums` array. The index `i` would normally start from `0`, but since we want to include `n` in the XOR operations, we start it from `1`.
3. The `reduce` function then takes two arguments: the first is the `xor` function, which specifies how the elements should be combined; and the second is the generator expression. What `reduce` does is, it applies the `xor` operation cumulatively to the elements of the iterable—sequentially applying the `xor` operation starting from the first pair to the last.
4. Since XOR is associative, the order of the application doesn't matter, and the `reduce` operation will effectively XOR all indices from 1 to `n` and all values in `nums`. The one number that isn't paired and XOR'd (the missing number) will remain.

The result of the `reduce` operation is the missing number from the array, which is returned by the `missingNumber` function. The beauty of this solution lies in its time complexity which is `O(n)` due to the single pass through the array, and space complexity which is `O(1)` since no extra space is used aside from the variables to perform the calculation.

Example Walkthrough

Let us take a small example to illustrate the solution approach. Suppose we are given an array `nums` as `[4, 2, 3, 0]`. Notice that in this case `n` is `4` as there are `5` unique numbers in the range [0-4], but we only have `4` numbers in the array. We should identify the missing number which is in the range `[0, 4]` but not in the array. Following the steps of the solution approach:

1. Generate pairs of index (starting from 1) and value from the array `nums`: The pairs will look like this `(index, value)` - `(1, 4)`, `(2, 2)`, `(3, 3)`, `(4, 0)`.
2. Compute the XOR of the index and value for each element:
   - The XOR of `(1, 4)` is `1 ^ 4 = 5`
   - The XOR of `(2, 2)` is `2 ^ 2 = 0`
   - The XOR of `(3, 3)` is `3 ^ 3 = 0`
   - The XOR of `(4, 0)` is `4 ^ 0 = 4`
3. Apply the `reduce` function over our pairs using the `xor` operation:
   - Start with the first pair resulting value, which is `5`.
   - Sequentially XOR this with the result of each of the subsequent pairs:
     - `5 ^ 0 = 5` (from second pair)
     - `5 ^ 0 = 5` (from third pair)
     - `5 ^ 4 = 1` (from fourth pair)
4. Finally, we are left with the value `1` which is the result of the cumulative XOR so far. To find the missing number, we need to also XOR `1` with `n` (`5` in this case because we should include `0` through `n`):
   - `1 ^ 5 = 4`

Now `4` is NOT the missing number; rather, it is the result of XORing all the indices with their respective values including the number `n`. Remember that by using the XOR operation, duplicate elements (the ones that exist in the array) should negate each other, leaving only the missing element. However, since we started the index from 1, we have not accounted for the `0`th index.

To fix this, we must one more time XOR our current result `4` with `0` (the missing 0th index due to starting enumeration at 1):

- `4 ^ 0 = 4`

At this point, we need to realize that we have XOR'd the number `4` twice: Once for the 0th index that was missing in our enumeration, and once for the value from the array `nums[0]`. Each number from `1` to `n` has been XOR'd twice except for the missing number, so these pairs have all canceled out, leaving us with just the missing number.

Thus, the missing number from the array `[4, 2, 3, 0]` is `1`, which is the missing element that did not get canceled out through the XOR operations. This is consistent with the range `[0, 4]` which contains `0, 1, 2, 3, 4` – every other number is present in the array except for `1`.

Python Solution

```python
from functools import reduce  # Import reduce function from functools module.
from operator import xor  # Import xor function from operator module.

class Solution:
    def missingNumber(self, nums):
        """
        Find the missing number in the array `nums` containing n distinct numbers in the
        range [0, n].

        Args:
        nums (List[int]): The input array with missing number.

        Returns:
        int: The missing number from the array.
        """
        # Apply the xor operator (^) between each number's index (i) and its value (v) starting with index 1.
        # The reduce function then applies the xor operation cumulatively to the pairs of elements,
        # which is needed since the input list misses one number, xor with the missing number results in the number itself.
        # The xor of a number with itself is 0, so it will leave out normal pairs and single out the missing value.
        # We enumerate starting with 1, since we want to include the number "n" in our comparisons.
        return reduce(xor, (i ^ v for i, v in enumerate(nums, 1)))

```

Java Solution

```java
class Solution {

    // This method finds the missing number in an array containing numbers from 0 to n
    public int missingNumber(int[] nums) {

        // Length of the array should be one number short of the full set
        int n = nums.length;
        // Initialize answer with the last number (which is n since array is 0-indexed)
        int result = n;

        // Iterating through the array
        for (int i = 0; i < n; ++i) {

            // Apply XOR operation between the current index and the element at that index, and XOR that with the current result
            // Since a number XORed with itself is 0 and a number XORed with 0 is the number itself, this will eventually leave us w
            result ^= i ^ nums[i];
        }

        // Return the result which is the missing number
        return result;
    }
}

```

C++ Solution

```cpp
#include <vector> // Include the vector header for using the vector class

class Solution {
public:
    int missingNumber(std::vector<int>& nums) {
        // Calculate the size of the vector.
        int n = nums.size();
        // Initialize 'missing' to the size of the vector, as the array is
        // supposed to contain all numbers from 0 to n, so 'n' is the initial candidate.
        int missing = n;

        // Loop through the array elements.
        for (int i = 0; i < n; ++i) {
            // XOR 'missing' with the current index 'i' and the element at that index 'nums[i]'.
            // The idea behind this is that when a number and its index are XOR'ed
            // in the range from 0 to n, the only number that will not be cancelled
            // out is the missing number.
            missing ^= (i ^ nums[i]);
        }

        // Return the missing number after processing the entire array.
        return missing;
    }
};

```

Typescript Solution

```typescript
function missingNumber(nums: number[]): number {
    // Calculate the length of the given array.
    const arrayLength = nums.length;

    // Initialize the result with the length of the array. This covers the edge case
    // where the missing number is exactly equal to the length of the array.
    let result = arrayLength;

    // Iterate through the array.
    for (let i = 0; i < arrayLength; ++i) {
        // XOR the current index with the current array element and the current result.
        // This will cancel out all numbers from 0 to n except the missing one.
        result ^= i ^ nums[i];
    }

    // Return the missing number once all numbers have been XOR'ed.
    return result;
}

```

Time and Space Complexity

The given Python code uses the `reduce` function and a generator expression to find the missing number in a list of unique integers ranging from `0` to `n`. Here's how we analyze the computational complexity:

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the array `nums`. This is because the generator expression `(i ^ v for i, v in enumerate(nums, 1))` iterates through each element of the array exactly once. The `xor` operation inside the generator expression is a constant-time operation, taking `O(1)` time. The `reduce` function then iterates through these `n` `xor` operations to combine them, which also takes `O(n)` time. So, the total time taken is linear in relation to the array's length.

Space Complexity

The space complexity of the code is `O(1)`, meaning it uses constant space regardless of the input size. The generator does not create an additional list in memory; it computes the `xor` values on-the-fly. The `reduce` function only needs space for the accumulator to hold the intermediate `xor` results, which is a single integer value. Thus the overall space used does not scale with the size of the input array.