260. Single Number III

Medium Bit Manipulation Array

Problem Description

of the elements appear exactly two times. The task is to find the two unique elements without using additional space and ensuring that the algorithm runs in linear time complexity. This means we cannot sort the array or use a data structure that would take up extra space proportional to the input size. The uniqueness of our solution should come from cleverly manipulating the numbers themselves to identify the two that are different from the rest.

Intuition

In this problem, we are given an integer array nums where precisely two elements are unique, appearing only once, and the rest

identical numbers, the result is zero; when we apply it to a number and zero, we get the original number back. These properties

are crucial: 1. If we XOR a number with itself, we get 0 $(x\oplus x=0)$. 2. Any number XORed with 0 remains unchanged $(x\oplus 0=x)$.

When we XOR all numbers in the array, we are left with the XOR of the two unique numbers, since all other numbers cancel each

To solve this problem efficiently, we should utilize the properties of the XOR (^) bitwise operation. When we apply XOR to two

other out (pairwise XORing them gives us 0).

Now, since the two unique numbers are distingue.

All duplicates will still cancel out within their respective groups.

Now, since the two unique numbers are distinct, their XOR result must have at least one bit set to 1, representing a place where they differ. If we can isolate this bit, we can divide all numbers in the array into two groups, based on whether they have this bit set to 1 or not. This step ensures that each group contains exactly one of the unique numbers, with all others still forming

cancelling pairs.

We obtain such a bit using xs & -xs, which isolates the lowest bit that is set to 1 in the XOR result.

With this bit as a mask, we iterate through all the numbers again, using the mask to split the numbers into two groups and performing XOR in each group to find the unique numbers. We are guaranteed that:

The two unique numbers will fall into separate groups and won't cancel out.
 The final result of the XOR operation in each group is the unique number within that group.

Solution Approach

The solution uses the bitwise XOR operation to pinpoint the two unique numbers in the array. The XOR operation is chosen for its

ability to cancel out pairs of identical numbers, meaning that XOR-ing all numbers in an array where each number appears twice,

We first apply XOR to all the elements in the array, which is conveniently done using the reduce() function with the xor

Since XOR-ing a number with itself results in 0, and any number XOR-ed with 0 is the number itself, all paired numbers cancel

except for two numbers, will leave us with the XOR of these two unique numbers. Here are the steps to this approach, using bitwise operations in Python:

each other, and we are left with:

xs = unique1 ^ unique2

for x in nums:

number.

Example Walkthrough

if x & lb:

a ^= x

operator from Python's functools module. The result of this operation is stored in variable xs and can be mathematically represented as:

xs = nums[0] ^ nums[1] ^ ... ^ nums[n - 1]

2. We need to find a way to separate unique1 and unique2. Since they are distinct, there must be at least one bit in which they differ. The line of code lb = xs & -xs finds the least significant bit that is set to 1 in the XOR result (xs). The -xs is a way to get the two's complement which flips all the bits of xs and adds 1, so when it's AND-ed with xs, all the bits are canceled out except for the least significant bit.

With this bit (1b), we divide the numbers into two groups and XOR the numbers in each group to find the unique numbers.

This code XORs all numbers that have the lb bit set. Since all other numbers appear twice, only unique1 or unique2 (whichever has that bit set) will be the result of this XOR operation (a holds this unique number).

4. To find the second unique number, XOR a with xs:
 b = xs ^ a
 Since xs is unique1 ^ unique2, by XOR-ing it with one unique number, the other is revealed, giving us b, the second unique

The final result, [a, b], contains the two numbers that occur only once in the array. The use of bitwise operations, along with

the loop through the array, ensures that the algorithm runs in linear time, O(n), with constant space complexity, O(1), as it

Let us take an array nums = [4, 1, 2, 1, 2, 3] to illustrate the solution approach.

We apply XOR to all elements:

 $xs = 4 ^1 ^2 ^3$

employs a fixed number of integer variables regardless of the input size.

each other, and we're left with the XOR of the two unique numbers ($xs = 4 ^ 3$).

lb = 7 & -7 # which isolates the least significant bit (0001)

This step is done by iterating through the array again with:

After applying XOR successively, we get:

xs = 4

This is because 1 ^ 1 and 2 ^ 2 both give 0, and XOR-ing 0 with any number yields that number. Thus, all duplicates cancel

a = 0
b = 0
for x in nums:

if x & lb:

Solution Implementation

from functools import reduce

from operator import xor

from typing import List

class Solution:

Python

Java

class Solution {

else:

Find the least significant bit that is set to 1:

 $xs = 4 ^ 3$ # which is 7 in binary (0111)

Upon iterating (4 has the least significant bit 0, 3 has it set to 1): $a = 3 \text{ (as } 1^1 = 0, 2^2 = 0, leaving 3)$

Since a holds one of the unique numbers, there is no need for the final XOR step to determine b. If we had not directly

The result 1b is 1, indicating the least significant bit that is different between the two unique numbers (4 and 3).

We now divide the numbers into two groups and XOR them separately based on the least significant bit set to 1:

a ^= x # This will be the group where the least significant bit is 1

b ^= x # This will be the group where the least significant bit is 0

b = 4 (as it is the only number remaining with least significant bit 0)

received the second unique number, we would XOR a with xs to find b:

Calculate the XOR of all the numbers, the result will be the XOR of

the two unique numbers as other numbers appear twice and thus cancel out.

Find the rightmost set bit in xor all nums by & with its two's complement.

// Initial bitmask value which will eventually contain the XOR of all numbers in the array

// Perform XOR across all elements in the array to find the XOR of the two unique numbers

// XOR the found unique number with xorResult to find the second unique number since

const xorOfUniqueNums = nums.reduce((accumulated, current) => accumulated ^ current);

// Iterate through all numbers to separate them into two groups and perform XOR.

// XOR the first unique number with xorOfUniqueNums to get the second unique number.

Calculate the XOR of all the numbers, the result will be the XOR of

the two unique numbers as other numbers appear twice and thus cancel out.

Find the rightmost set bit in xor all nums by & with its two's complement.

This bit will be different in the two unique numbers, so we can use it

Loop through all numbers and XOR those with the rightmost set bit.

This isolates one of the unique numbers due to the bitwise & filter.

const rightmostSetBit = xorOfUniqueNums & -xorOfUniqueNums;

// The groups are based on whether they have the rightmost set bit.

// XOR numbers that have the rightmost set bit.

def singleNumber(self, nums: List[int]) -> List[int]:

to distinguish between the two groups.

if num & rightmost set bit:

rightmost_set_bit = xor_all_nums & -xor_all_nums

xor all nums = reduce(xor, nums)

// The result will be one of the unique numbers.

let firstUniqueNumber = 0;

for (const num of nums) {

from functools import reduce

unique num 1 = 0

for num in nums:

the input array.

from operator import xor

from typing import List

class Solution:

if (num & rightmostSetBit) {

firstUniqueNumber ^= num;

// Get the rightmost set bit in xorOfUniqueNums. This bit will be set in one unique number and not in the other.

This bit will be different in the two unique numbers, so we can use it

Find the second unique number by XORing the first unique number

with xor all nums (which is the XOR of the two unique numbers).

// XOR the numbers that have the set bit (same as in rightmostSetBit)

 $\# b = xs \land a$ (only necessary if b was not directly received from the loop)

After the loop, a holds the value 3, and b holds the value 4.

def singleNumber(self, nums: List[int]) -> List[int]:

to distinguish between the two groups.

if num & rightmost set bit:

unique_num_1 ^= num

return [unique_num_1, unique_num_2]

public int[] singleNumber(int[] nums) {

unique_num_2 = xor_all_nums ^ unique_num_1

int rightmostSetBit = xorResult & -xorResult;

if ((num & rightmostSetBit) != 0) {

// xorResult is a XOR of both unique numbers

// Return an array containing both unique numbers

int secondUniqueNumber = xorResult ^ firstUniqueNumber;

return new int[] {firstUniqueNumber, secondUniqueNumber};

firstUniqueNumber ^= num;

Return the list containing the two unique numbers.

xor_all_nums = reduce(xor, nums)

The final result is [3, 4], which contains the two unique numbers from the array. This example illustrates how the bitwise XOR and AND operations can be used to solve the problem efficiently, adhering to both time and space complexity constraints.

rightmost_set_bit = xor_all_nums & -xor_all_nums

unique num 1 = 0

Loop through all numbers and XOR those with the rightmost set bit.

This isolates one of the unique numbers due to the bitwise & filter.

for num in nums:

// Get the rightmost set bit in xorResult which will differentiate the two unique numbers // Negating xorResult gives its two's complement, and bitwise AND with original number // isolates the rightmost set bit

int xorResult = 0:

for (int num : nums) {

xorResult ^= num;

int firstUniqueNumber = 0;

for (int num : nums) {

// to find one of the unique numbers

// Check if the bit is set

C++ #include <vector> class Solution { public: std::vector<int> singleNumber(std::vector<int>& nums) { // Initial total XOR value for all numbers in the array long long total xor = 0; for (int num : nums) { total_xor ^= num; // Find the rightmost set bit in the total XOR (first bit that differs between the two unique numbers) int last bit = total xor & -total xor; int num1 = 0; for (int num : nums) { // Separate numbers into two groups and find the first unique number by XORing numbers in the same group if (num & last bit) { num1 ^= num; // The second unique number is found by XORing the first unique number with the initial total XOR int num2 = total_xor ^ num1; // Return the two unique numbers return {num1, num2}; **}**; **TypeScript** function singleNumber(nums: number[]): number[] { // Perform XOR on all numbers to get the XOR of the two unique numbers.

const secondUniqueNumber = xorOfUniqueNums ^ firstUniqueNumber; // Return the two unique numbers. return [firstUniqueNumber, secondUniqueNumber];

Find the second unique number by XORing the first unique number
with xor all nums (which is the XOR of the two unique numbers).
unique_num_2 = xor_all_nums ^ unique_num_1

Return the list containing the two unique numbers.
return [unique_num_1, unique_num_2]

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 there is a single loop in the

For the space complexity, it is 0(1) which signifies constant space usage. Outside of the input array, the code only uses a fixed

number of integer variables (xs, a, lb, b), irrespective of the input size, hence the space used does not scale with the size of

function that iterates over all the elements in the array exactly once to find the numbers that appear just once.