Introduction

We'll cover the following

- What could go wrong with the above algorithm?
- Important properties of XOR to remember

XOR is a logical bitwise operator that returns 0 (false) if both bits are the same and returns 1 (true) otherwise. In other words, it only returns 1 if exactly one bit is set to 1 out of the two bits in comparison.

A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

It is surprising to know the approaches that the XOR operator enables us to solve certain problems. For example, let's take a look at the following problem:

> Given an array of $n - 1$ integers in the range from 1 to n, find the one number that is missing from the array.

Example:

```
Input: 1, 5, 2, 6, 4
Answer: 3
```

A straight forward approach to solve this problem can be:

1. Find the sum of all integers from 1 to n; let's call it `s1`.
2. Subtract all the numbers in the input array from `s1`; this will give us the missing number.

This is what the algorithm will look like:

Java | Python3 | JS | C++

```python
def find_missing_number(arr):
  n = len(arr) + 1
  # find sum of all numbers from 1 to n.
  s1 = 0
  for i in range (1, n+1):
    s1 += i

  # subtract all numbers in input from sum.
  for i in arr:
    s1 -= i

  # s1, now, is the missing number
  return s1

def main():
  arr = [1, 5, 2, 6, 4]
  print('Missing number is:' + str(find_missing_number(arr)))
```

```
main()
```

Run Save Reset

Time & Space complexity: The time complexity of the above algorithm is $O(n)$ and the space complexity is $O(1)$.

What could go wrong with the above algorithm?

> While finding the sum of numbers from 1 to n, we can get integer overflow when n is large.

How can we avoid this? Can XOR help us here?

Remember the important property of XOR that it returns 0 if both the bits in comparison are the same. In other words, XOR of a number with itself will always result in 0. This means that if we XOR all the numbers in the input array with all numbers from the range 1 to n then each number in the input is going to get zeroed out except the missing number. Following are the set of steps to find the missing number using XOR:

1. XOR all the numbers from 1 to n, let's call it `x1`.
2. XOR all the numbers in the input array, let's call it `x2`.
3. The missing number can be found by `x1 XOR x2`.

Here is what the algorithm will look like:

Java Python3 JS C++

```python
def find_missing_number(arr):
  n = len(arr) + 1
  # x1 represents XOR of all values from 1 to n
  x1 = 1
  for i in range(2, n+1):
    x1 = x1 ^ i

  # x2 represents XOR of all values in arr
  x2 = arr[0]
  for i in range(1, n-1):
    x2 = x2 ^ arr[i]

  # missing number is the xor of x1 and x2
  return x1 ^ x2

def main():
  arr = [1, 5, 2, 6, 4]
  print('Missing number is:' + str(find_missing_number(arr)))

main()
```

Run Save Reset

Time & Space complexity: The time complexity of the above algorithm is $O(n)$ and the space complexity is $O(1)$. The time and space complexities are the same as that of the previous solution but, in this algorithm, we will not have any integer overflow problem.

Important properties of XOR to remember

Following are some important properties of XOR to remember:

- Taking XOR of a number with itself returns 0, e.g.,
  - 1 ^ 1 = 0
  - 29 ^ 29 = 0
- Taking XOR of a number with 0 returns the same number, e.g.,
  - 1 ^ 0 = 1
  - 31 ^ 0 = 31
- XOR is Associative & Commutative, which means:
  - (a ^ b) ^ c = a ^ (b ^ c)
  - a ^ b = b ^ a

In the following chapters, we will apply the XOR pattern to solve some interesting problems.

← Back

Solution Review: Problem Challenge 3

Next →

Single Number (easy)

Completed

Report an Issue Ask a Question