Topic Introduction

Today we're diving into an essential array interview technique: **"Finding the Missing or Duplicate Number."**

This is a classic pattern that shows up in many forms during interviews. The core idea? Given an array of numbers drawn from a certain range, you're asked to spot what's missing—or what's there twice. This usually involves using mathematical properties, clever in-place tricks, or bit manipulation.

What is this pattern?

Imagine you have an array of numbers, but you know something is off: maybe one number is missing, maybe one is a duplicate, or maybe you need to find the smallest absent positive integer. The challenge is to do this efficiently, often without extra space.

How does it work?
The key is harnessing information about the numbers' range and leveraging properties like sums, indices, or cyclic placement.

- Sometimes, you compare the expected sum to the actual sum.
- Sometimes, you use array indices as markers.
- Sometimes, you rearrange the array in-place.

Why is this useful?
These problems test your understanding of array manipulation, constraints, and efficiency. Interviewers love them because they uncover your approach to both brute force and optimization.

Quick Example (not from the main problems):
Suppose you have `[3, 0, 1]`, and you know the numbers should be from 0 to 3. One is missing—can you find it? (Hint: Sum should be 6, but the array sums to 4, so 2 is missing!)

The Three Problems for Today

Let's look at three popular LeetCode problems that use this pattern:

- **Missing Number** (link): Find the one number missing from `[0, n]`.
- **First Missing Positive** (link): Find the smallest missing positive integer in an unsorted array.
- **Find the Duplicate Number** (link): In an array of `n+1` numbers from `1` to `n`, one number is present twice; find it.

Why these together?
They all ask you to spot the "odd one out"—missing or duplicated numbers in arrays. The challenge lies in constraints: sometimes you can use extra space, sometimes not; sometimes the numbers start at 0, sometimes at 1; sometimes you're looking for a missing, sometimes a duplicate.

We'll start with the simplest, then ramp up the challenge!

Problem 1: Missing Number

Problem Statement (Rephrased):

Given an array `nums` containing `n` distinct numbers from `0` to `n`, find the one number that is missing from the range.

[LeetCode 268: Missing Number]

Example:

Input: `nums = [3, 0, 1]`

Output: `2`

Explanation: Numbers from 0 to 3 are [0,1,2,3]. 2 is missing.

Try this one yourself:

Input: `nums = [0, 1]`

What's missing?

Brute Force Approach

The slowest way:

- For each number from 0 to n, check if it's in the array.
- Time: O(n^2) (because for each number, you scan the array).

Optimal Approach: Sum Formula

Let's use the pattern!

- The sum of numbers from 0 to n is `n*(n+1)//2`.
- Subtract the sum of the array from this expected sum—the difference is the missing number.

Step-by-step Breakdown:

- Calculate expected sum: `n*(n+1)//2`
- Calculate actual sum: `sum(nums)`
- The missing number is `expected - actual`

Python Solution

```
def missingNumber(nums):
    n = len(nums)
    expected_sum = n * (n + 1) // 2  # Sum of numbers from 0 to n
    actual_sum = sum(nums)           # Sum of given numbers
    return expected_sum - actual_sum # Missing number is the difference

# Example usage:
nums = [3, 0, 1]
print(missingNumber(nums))  # Output: 2
```

Time Complexity: O(n)

Space Complexity: O(1) (Only a few extra variables)

How does it work?

- `expected_sum` gives what the total *should* be if nothing was missing.
- `actual_sum` is what you have.
- Subtract to find what's missing.

Trace for [3, 0, 1]:

- n = 3
- expected_sum = 3*4/2 = 6
- actual_sum = 3+0+1 = 4
- missing = 6 - 4 = 2

Try this test case yourself:

`nums = [9,6,4,2,3,5,7,0,1]`

What's missing?

Take a moment to solve this on your own before jumping into the solution!

Did you know?

You can also solve this with bitwise XOR! Try it out after you master the sum formula approach.

Problem 2: First Missing Positive

[LeetCode 41: First Missing Positive]

Problem Statement (Rephrased):

Given an unsorted array, find the smallest missing positive integer (greater than 0).

Example:

Input: `[3, 4, -1, 1]`

Output: `2`

Explanation: The numbers present are 1, 3, 4. 2 is the smallest missing positive.

How is this different?

- Only care about **positive** numbers.
- The array is unsorted and may have negatives and duplicates.
- You need to find the **smallest** missing, not just any missing.

Try this one:

Input: `[1, 2, 0]`

What's the smallest missing positive?

Brute Force Approach

- Check for 1, 2, 3,... up to n+1; for each, scan the array to see if it exists.
- O(n^2), not efficient.

Optimal Approach: In-Place "Cyclic Sort" Pattern

Here's the trick:

- Since the smallest missing positive must be between 1 and n+1, try to place each number in its "correct" position:
  - 1 at index 0, 2 at index 1, ..., n at index n-1
- Swap numbers into their correct positions as long as they are in [1, n] and not already in place.

Step-by-step Breakdown:

- For each index i:
    - While nums[i] is in [1, n] and not already in the right spot, swap nums[i] with nums[nums[i]-1]
- After this, scan the array:
    - The first index i where nums[i] != i+1, return i+1
- If all positions are correct, return n+1

Pseudocode

```
for i from 0 to n-1:
    while nums[i] in 1..n and nums[nums[i]-1] != nums[i]:
        swap nums[i] and nums[nums[i]-1]

for i from 0 to n-1:
    if nums[i] != i+1:
        return i+1

return n+1
```

Example Trace for [3, 4, -1, 1]:

- Initial: [3, 4, -1, 1]
- Place 3 at index 2: swap nums[0] with nums[2] -> [-1, 4, 3, 1]
- Now, -1 is out of range, move on.
- 4 at index 1: swap nums[1] with nums[3] -> [-1, 1, 3, 4]
- 1 at index 1: swap nums[1] with nums[0] -> [1, -1, 3, 4]
- Done swapping.
- Scan: index 0: 1, index 1: -1 (should be 2), so return 2.

Test case for you:

Input: `[7, 8, 9, 11, 12]`

What's the answer?

Time Complexity: O(n)

Space Complexity: O(1) (in-place)

Problem 3: Find the Duplicate Number

[LeetCode 287: Find the Duplicate Number]

Problem Statement (Rephrased):

Given an array of `n+1` integers where each integer is between 1 and n (inclusive), and only one value is repeated, find the duplicate number. You cannot modify the array and must use constant extra space.

How is this different?

- Now, there is a duplicate, not a missing number.
- The array has n+1 numbers with values between 1 and n.
- You cannot change the array!

Try this:

Input: `[1, 3, 4, 2, 2]`

What's the duplicate?

Brute Force

- For each element, count occurrences (O(n^2))
- Or, use a set to track seen numbers (O(n) time, O(n) space)

Optimal: Floyd's Tortoise and Hare (Cycle Detection)

Why?

Since values are in [1, n], you can treat the array as a linked list where each index points to the value at that index.

A duplicate means there's a cycle!

Pseudocode

```
# Phase 1: Find intersection point
slow = nums[0]
fast = nums[0]
do:
    slow = nums[slow]
    fast = nums[nums[fast]]
```

```
while slow != fast

# Phase 2: Find entrance to cycle
slow = nums[0]
while slow != fast:
    slow = nums[slow]
    fast = nums[fast]

return slow
```

Example for [1, 3, 4, 2, 2]:

- slow and fast start at nums[0] = 1
- slow = nums[1]=3, fast=nums[nums[1]]=nums[3]=2
- Continue until they meet. Then, reset slow to start and move both one step at a time—the meeting point is the duplicate.

Test case for you:
Input: `[3, 1, 3, 4, 2]`
What's the duplicate?

Time Complexity: O(n)
Space Complexity: O(1)

Summary and Next Steps

Today, you learned to:

- Spot *missing* and *duplicate* numbers in arrays.
- Use the sum formula, in-place cyclic sort, and cycle detection.
- Recognize how array constraints open the door to clever techniques.

Key Patterns to Remember:

- If the array has a known range, use the sum or XOR trick.
- If missing positive, try to place numbers at their "home" indices.
- If duplicates but can't modify or use extra space, think of cycle detection.

Common Pitfalls:

- Off-by-one errors with ranges (0 to n vs 1 to n)
- Forgetting to handle negatives or out-of-range numbers
- Accidentally modifying the array when forbidden

Action List

- **Solve all three problems on your own**, even if you've seen the code.

- **Try different approaches:**
  - Use XOR to solve the Missing Number problem.
  - Use a set to solve Find the Duplicate Number and compare to the cycle detection approach.
- **Dry-run with pen and paper** for tricky in-place swaps or cycle detection.
- **Seek out similar problems:**
  - E.g., “Find All Duplicates,” “Find All Missing Numbers.”
- **Compare edge-case handling** with other people’s solutions.
- **Don’t stress if you get stuck!** The more you practice, the more these patterns become second nature.

Keep at it! Every “aha!” moment sharpens your interview skills.