Topic Introduction

Today, we'll dig into the clever world of **bit manipulation** — a small but mighty toolkit for coding interviews. Bit manipulation means working directly with the binary bits of numbers, using bitwise operators like AND (`&`), OR (`|`), XOR (`^`), NOT (`~`), left shift (`<<`), and right shift (`>>`).

What is bit manipulation?

It's the art of using these bitwise operators to solve problems in a way that's often faster and more memory-efficient than using arrays or hash maps. With bit manipulation, you're thinking like a computer: flipping, masking, or comparing the 0s and 1s that make up each number.

How does it work?

Imagine the number 5. In binary, it's `101`. With bit manipulation, you could flip its bits, check if it's even or odd, or find patterns in a list of numbers — all with just a few operations.

When is it useful in interviews?

Bit manipulation shines when you need to:

- Find unique elements (e.g. every number appears twice except one)
- Work with sets of flags
- Optimize for space and speed
- Solve problems involving parity, power-of-two checks, or toggling values

Simple Example:

Let's say you want to check if a number is odd or even.

- If the last bit is 1, it's odd; if 0, it's even.
- So, `num & 1` is 1 for odd, 0 for even.

```
num = 7
print(num & 1)  # Output: 1 (odd)
```

Now, let's jump into a trio of classic interview problems that use bit manipulation to find unique elements in arrays.

Why These 3 Problems?

We're grouping together:

- Single Number
- Single Number II
- Single Number III

because each one challenges you to find unique numbers in an array using bitwise tricks:

- **Single Number:** Find the lone element in an array where every other number appears *exactly twice*.
- **Single Number II:** Every number appears *three times* except for one unique element.
- **Single Number III:** Every number appears *twice* except for *two* unique elements.

They escalate in complexity, but all use bit manipulation for efficient solutions.

Problem 1: Single Number

Problem link: Single Number (LeetCode 136)

Rephrased Statement:

You're given an array of integers where every element appears *twice* except for one. Find that single element.

Example:

Input: `[2, 2, 1]`

Output: `1`

Because 2 appears twice, and 1 appears just once.

Walkthrough:

Imagine writing down all numbers. You'd cross out each pair, and the single number would be left. But how do you do this efficiently?

Try this input:

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

Output: `4`

Brute-force Approach:

- For each number, count how many times it appears.
- Return the number that appears once.
- **Time:** O(n^2) (slow, uses nested loops)
- **Space:** O(1) (no extra storage)

Optimal Approach:

Let's use **XOR (^)**. It's magical for this problem:

- Any number XOR itself is 0 (`a ^ a = 0`)
- Any number XOR 0 is itself (`a ^ 0 = a`)
- XOR is commutative and associative (order doesn't matter)

So, XOR-ing all numbers together cancels out pairs, leaving the single number.

Step-by-step:

- Start with `result = 0`.
- For each number in array: `result = result ^ num`.
- Return `result`.

Python Solution:

```
def singleNumber(nums):
    result = 0
    for num in nums:
        result ^= num  # XOR each number; pairs cancel out
    return result
```

- **Time Complexity:** O(n) (linear, one pass)
- **Space Complexity:** O(1) (constant space)

Code Breakdown:

- `result = 0`: Start with zero, since XOR with zero does nothing.
- `result ^= num`: XORs `result` with each number. Pairs zero out; the unique number remains.
- `return result`: Returns the single number.

Trace Example:

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

- Start: result = 0
- result ^ 4 = 4
- 4 ^ 1 = 5
- 5 ^ 2 = 7
- 7 ^ 1 = 6
- 6 ^ 2 = 4

Final answer: 4

Try this input yourself:

Input: `[7, 3, 5, 3, 7]`

What's the output?

Take a moment to solve this on your own before checking the code!

Reflect:

Did you know this could also be solved using a hash map? Try that after you master the XOR approach!

Problem 2: Single Number II

Problem link: Single Number II (LeetCode 137)

Rephrased Statement:

Given an array where every element appears *three times* except one, find that unique element.

What's similar?

Still finding a unique element, but now every other number appears three times, not two. The XOR trick from before doesn't work directly.

Brute-force:

- Count occurrences of each number (use a hash map)
- Find the one with count 1
- **Time:** O(n)
- **Space:** O(n) (uses extra space)

Optimal Approach:

Let's use **bitwise counting**.

Core Idea:

- For each of the 32 bits (for a 32-bit integer), count how many times that bit is set (is a 1) across all numbers.
- If a bit is set in the unique number, the total count for that bit won't be a multiple of 3.
- For each bit, if `count % 3 != 0`, set that bit in the result.

Step-by-step:

- Initialize result = 0.
- For each bit position (0 to 31):
    - Count how many numbers have this bit set.
    - If count % 3 != 0, set that bit in result.
- Return result.

Example:

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

- 2 appears three times, 3 appears once.
- Output: `3`

Pseudocode:

```
result = 0
for i from 0 to 31:
    count = 0
    for num in nums:
        if (num >> i) & 1:
            count += 1
    if count % 3 != 0:
        result |= (1 << i)
# Handle negative numbers if needed
if result >= 2**31:
    result -= 2**32
return result
```

Trace Example:

For `[2, 2, 3, 2]`:

- For bits 0, 1: count how many times set.
- Only bit 0 is set in 3 but not in 2.
- Result is 3.

Try this input yourself:

Input: `[0, 1, 0, 1, 0, 1, 99]`

What's the output?

Time Complexity: O(32n) = O(n)

Space Complexity: O(1)

Problem 3: Single Number III

PrepLetter: Single Number and similar

Problem link: Single Number III (LeetCode 260)

Rephrased Statement:

Given an array where every element appears *twice* except for *two* unique elements, find those two elements.

What's different?

Now, there are *two* unique numbers! We want to find both.

Brute-force:

- Use a hash map to count occurrences.
- Collect all numbers with count 1.
- **Time:** O(n)
- **Space:** O(n)

Optimal Approach:

Let's use an **XOR partition trick**:

Key Insight:

- XOR all numbers. The result is `a ^ b` where a and b are the two unique numbers.
- Find any bit that is set in `a ^ b` (means a and b differ at that bit).
- Partition numbers into two groups: those with that bit set, and those without.
- XOR numbers in each group — each group will give you one unique number!

Pseudocode:

```
xor_all = 0
for num in nums:
    xor_all ^= num   // xor_all = a ^ b

# Find rightmost set bit
diff = xor_all & -xor_all

a = 0
b = 0
for num in nums:
    if num & diff:
        a ^= num
    else:
        b ^= num
return [a, b]
```

Example:

Input: `[1, 2, 1, 3, 2, 5]`

- XOR all: 1^2^1^3^2^5 = 3^5 = 6
- Rightmost set bit: diff = 2
- Partition: numbers with bit 2 set: [2,2,3]; without: [1,1,5]
- XOR in each group: 3, 5

Try this input yourself:

Input: `[4, 1, 2, 1, 2, 5]`

What are the two unique numbers?

Time Complexity: O(n)

Space Complexity: O(1)

Can you see how the partitioning allows us to isolate each unique number?

Summary and Next Steps

We grouped these problems because they're all classic examples of using **bit manipulation** to find unique elements in arrays of duplicates — but each adds its own twist.

Key Patterns to Remember:

- XOR is your friend when every element appears twice except one.
- Bit counting (modulo group size) helps when every element appears k times except one.
- XOR partitioning can isolate two unique numbers among pairs.

Common Mistakes:

- Forgetting to handle negatives in bit counting (for Python/Java).
- Misapplying XOR tricks when the group size isn't two.
- Overcomplicating with extra space when a neat bitwise solution exists.

Action List:

- Solve all three problems yourself, even the one with code here.
- Try a hash map approach for each — compare speed and space.
- Explore other bit manipulation problems (e.g., finding a missing number, counting bits).
- Review edge cases (empty array, negative numbers, large numbers).
- Compare your code with others — style and edge-case handling matter.
- If you get stuck, don't worry — keep practicing, and these patterns will become second nature!

Happy coding — and may your bits always fall into place!