🧠 Bit Manipulation Interview Quick-Check Pattern

A concise and powerful guide to solving bit-related problems in coding interviews.

---

1. 🧠 When to Use Bit Manipulation?

Ask yourself:

- ❓ Is the problem dealing with **binary representation**?
- ❓ Do I need to find a **unique element**, **power of 2**, or **bit count**?
- ❓ Are we optimizing for **space** or avoiding extra data structures?
- ❓ Is the input space **huge**, but the values are **small integers**?
- ❓ Are we dealing with **states**, **flags**, or **parity**?

> 💡 Bit manipulation is often used for **speed, space, or clever math**.

---

2. 🔧 Common Bit Tricks & Patterns

Operation	Code	Use Case
Check if `i`-th bit is set	`(num >> i) & 1`	Test a specific bit
Set the `i`-th bit	`num	(1 << i)`	Disable a bit
Unset the `i`-th bit	`num & ~(1 << i)`	Disable a bit
Toggle the `i`-th bit	`num ^ (1 << i)`	Flip a bit
Is number power of 2	`n > 0 && (n & (n - 1)) === 0`	Power of 2 check
Count set bits	`n &= (n - 1)` in loop	Brian Kernighan's Algorithm
XOR all elements	`res ^= num`	Find non-duplicate element

---

3. 🧪 Must-Know Bit Problem Templates

✅ Find Single Number (XOR)

```
function singleNumber(nums) {
  let result = 0;
  for (let num of nums) {
    result ^= num;
  }
  return result;
}
```

✅ Check Power of Two

```
function isPowerOfTwo(n) {
  return n > 0 && (n & (n - 1)) === 0;
}
```

✅ Count 1 Bits (Hamming Weight)

```
function hammingWeight(n) {
  let count = 0;
  while (n !== 0) {
    count++;
    n &= (n - 1);
  }
  return count;
}
```

✅ Sum Without + or -

```
function getSum(a, b) {
  while (b !== 0) {
    let carry = a & b;
    a = a ^ b;
    b = carry << 1;
  }
  return a;
}
```

✅ Find Missing Number (XOR from 0 to n)

```
function missingNumber(nums) {
  let xor = 0;
  for (let i = 0; i < nums.length; i++) {
    xor ^= i ^ nums[i];
  }
  return xor ^ nums.length;
}
```

---

4. 🧱 Edge Cases to Watch For

- Negative numbers (especially for signed/unsigned shifts)
- Zero and one (base cases in bit problems)
- Overflow/underflow in 32-bit integers
- JavaScript's behavior: numbers are 64-bit floats but bitwise ops use 32-bit ints
- Left shifting too far: `(1 << 31)` may go negative
- Infinite loop if bit clearing isn't done properly

🧠 When debugging, print `.toString(2)` to see the actual bits.

---

5. 🧠 Mental Model for Bit Problems

Core Trigger Phrases:

If the question says…	Think about using…
"Every element appears twice"	XOR
"Exactly one unique number"	XOR
"Bits", "Binary", "Flags"	Shift, AND, OR, XOR
"No extra space"	Bit tricks instead of sets/maps
"Count number of 1s"	Bit mask or Brian Kernighan's
"Power of 2"	`n & (n - 1)` trick

🔁 Problem Solving Loop

1. ❓ Is the problem fundamentally about **binary logic**?
2. 🔄 Can XOR/AND/OR replace conditionals or hash maps?
3. 🧠 Can I use shifting instead of loops or arrays?
4. ✂️ Can I isolate or manipulate specific bits?
5. 💡 Any **math shortcuts** using properties like `n & (n - 1)`?

✅ Final Interview Checklist

- Did I consider using XOR for uniqueness problems?
- Can the problem be solved with **bit masking** instead of extra space?
- Am I handling **0**, **1**, and **negative values** correctly?
- Did I test the output with `.toString(2)` for binary visualization?
- Have I checked if this needs **bit manipulation for optimization**?