Master Bit Manipulation

🔥 1. Understand the Big Picture: Bit Manipulation = Math + Patterns

Forget the idea that it's about tricks.

Bit manipulation is about:

- **Binary Math**
- **Patterns in data**
- **State representation using bits**

💡 Mindset shift: "I'm not learning tricks. I'm learning a new way to represent logic using bits."

🧠 2. Master the 7 Core Bit Concepts

I'd commit to learning and **implementing all of these** until I can use them **blindfolded**:

Concept	Practice Ideas
1. Get/Set/Clear/Toggle ith bit	Use `(1 << i)` patterns
2. Count set bits	Loop, Kernighan's algo, `Integer.bitCount()`
3. Check power of 2	`(n & (n - 1)) == 0`
4. XOR properties	Single number, Swap without temp, etc.
5. Rightmost set bit	`n & -n`
6. Bitmasking	Subsets, toggles, visited states
7. Shift operations	`<<`, `>>`, multiply/divide by 2

For each one, I'd solve **2–3 small problems** manually (on paper or console) and **visualize** the binary form.

📚 3. Solve Problems in Progressive Categories

I would **not** go to LeetCode randomly.

Instead, I'd solve in this order — just like top CP (competitive programming) people do:

Level	Category	Sample Problems
🟢 Easy	Get/set/toggle bits	“Get ith bit”, “Is even/odd”, “Clear ith bit”
🟡 Medium	XOR logic	“Single Number”, “Missing Number”, “Two Non-repeating Elements”
🟠 Medium+	Bitmasking	“Generate All Subsets”, “TSP DP with bitmask”
🔴 Hard	Advanced bit math	“Single Number II (thrice)”, “AND of range”, “Count bits in 1 to N”

I'd treat this like a **ladder**. Solve 3–5 problems per category and **write the thought process**, not just code.

🎯 4. Force myself to think in BINARY

Before solving any bit problem, I would:

- Convert all numbers to binary (on paper or console)
- Simulate each step by hand (even shifting, XOR, etc.)

This builds **intuition** and breaks fear.

🛠️ 5. Build Visual Tools (or use existing)

If I were serious, I’d either:

- Build a **bit visualizer** (takes an int, shows bits, tracks shifts)
- Or use sites like Visualgo.net, [pythontutor.com], or make a spreadsheet that lets me **see binary** change live

Visualization turns mystery into mastery.

✍️ 6. Make a Cheatsheet (from ME, not Google)

I'd write my own one-pager:

- Every operator with example
- Every pattern with diagram
- One-liner snippets for power of 2, get ith bit, etc.

Because **what you write, you remember**.

I'd revise this every 2–3 days for 2 weeks.

🔁 7. Apply to Real Use-Cases

Top companies love:

- **Subsets with bitmasking** (e.g., Google, Meta)
- **DP with bitmask** (e.g., TSP, assignment problems)
- **Optimization using bits** (e.g., avoid extra space)
- **Bit tricks to reduce time complexity**

I'd take problems I already solved and try to optimize them with bit manipulation — like:

- Replace array or map with bitmask
- Represent visited states using int/bitmask

💻 8. Push to GitHub, Teach Others

I'd post:

- My bit manipulation guide
- Solved problems with comments
- Diagrams

Maybe even **YouTube shorts/Reels** teaching:

> "Power of 2 in 30 seconds"

Why?

> **Teaching is the final stage of mastery.**

🧘 9. Mentally prepare for weird questions

Bit problems in interviews often look strange at first.

Example:

> “Given a range [m, n], find the bitwise AND of all numbers in the range.”

Looks scary — but becomes easy if I visualize.

So I’d **stay calm**, draw binary, and find patterns.

🔟 10. Reflect & Repeat

Every week, I’d review:

- Which concepts I’m weak in
- Which problems I couldn’t solve
- Which tricks I understood after solving

Goal: Within 4–6 weeks, bit manipulation becomes **second nature** — like solving puzzles.

🧠 Summary: If I Were a Human...

I’d **stop memorizing**, start:

1. Understanding **core patterns**
2. Practicing **problem types**, not problems
3. Writing **my own notes and tools**
4. Applying bit logic to **optimize code**
5. Explaining concepts to others

Problems for Practice

✅ Beginner Level — Core Bit Tricks

1. Get/Set/Clear/Toggle ith Bit

- ✅ *Problem:* Write functions for each bit operation.
- 🔸 *Example:* `n = 5 (0101)`, `i = 1`
  - Get → `0`
  - Set → `7 (0111)`
  - Clear → `5 (0101)`
  - Toggle → `7 (0111)`

---

2. Check if Number is Power of 2

- 🔸 *Input:* `n = 8`
- 🔸 *Output:* `true`
- 📌 Pattern: `(n & (n-1)) == 0`

---

3. Count Set Bits

- 🔸 *Input:* `n = 13 (1101)`
- 🔸 *Output:* `3`
- 📌 Pattern: Loop while `n != 0`, use `n & 1`

---

4. Rightmost Set Bit

- 🔸 *Input:* `n = 12 (1100)`
- 🔸 *Output:* `4 (0100)`
- 📌 Pattern: `n & -n`

🟡 Intermediate Level — XOR Logic

5. Find the Single Number (others twice)

(LeetCode #136)

- 🔸 *Input:* `[2, 3, 5, 3, 2]`
- 🔸 *Output:* `5`
- 📌 XOR all elements

---

6. Find Two Non-Repeating Numbers

(GFG version)

- 🔸 *Input:* `[2, 4, 7, 9, 2, 4]`
- 🔸 *Output:* `7, 9`
- 📌 XOR + bit partition

---

7. Missing Number from 0 to N

(LeetCode #268)

- 🔸 *Input:* `[3, 0, 1]`
- 🔸 *Output:* `2`
- 📌 XOR all indices + array elements

🟠 Advanced Level — Bitmasking & Count Tricks

8. Single Number II (others thrice)

(LeetCode #137)

- 🔸 *Input:* `[2, 2, 3, 2]`
- 🔸 *Output:* `3`
- 📌 Count bits in each position

---

9. Generate All Subsets of Array

(LeetCode #78)

- 🔸 *Input:* `[1, 2]`
- 🔸 *Output:* `[[], [1], [2], [1,2]]`
- 📌 Use bitmask of size `2^n`

---

10. Bitwise AND of Numbers Range [m, n]

(LeetCode #201)

- 🔸 *Input:* `m = 5, n = 7`
- 🔸 *Output:* `4`
- 📌 Left shift until m == n

---

11. Counting Bits for All Numbers ≤ N

(LeetCode #338)

- 🔸 *Input:* `n = 5`
- 🔸 *Output:* `[0,1,1,2,1,2]`
- 📌 `res[i] = res[i >> 1] + (i & 1)`

🔴 Bonus Challenges

12. Max AND Value of Any Pair

- 🔸 *Input:* `[4, 8, 6, 2]`
- 🔸 *Output:* `4`
- 📌 Try all pairs, track max `(a & b)`

---

13. TSP Using Bitmask + DP

- 🔸 *Input:* Distances between cities
- 🔸 *Output:* Minimum tour cost
- 📌 Use `dp[mask][i]` approach

- 🎯 *Hard but great real-world example*