Topic Introduction

Today, we're diving into a classic family of interview questions: **detecting if a number is a power of a specific base**. This pattern shows up frequently, often disguised with a bit of twist, and is a fantastic way to demonstrate mastery of math, loops, and bit manipulation.

What does "power of X" mean?

A number n is a power of x if it can be written as x^k for some integer k >= 0. For example, 16 is a power of 2 because 16 = 2^4. The concept is fundamentally about repeated multiplication.

How does this idea work in coding?

To check if a number is a power of some base, you typically need to see whether you can repeatedly divide by that base and end up at exactly 1, without leftovers. Alternatively, for certain bases, clever bit tricks can make things even faster.

Why do interviewers love this?

- **Core math + code:** It tests your ability to translate mathematical definitions into efficient code.
- **Tricky edge cases:** Handling 0, negatives, and big numbers is essential.
- **Optimizations:** Can you recognize when a brute force loop is enough, or when bit manipulation is better?

Simple Example (not one of our main problems)

Suppose we want to check if 27 is a power of 3. Start with 27, divide by 3 repeatedly:
27 -> 9 -> 3 -> 1.
Since we land on 1, and never get a remainder, 27 is a power of 3.

The Power Family

Today's problems are all about determining if a number is a power of 2, 3, or 4. Each uses a similar core idea, but with some unique coding twists:

- Power of Two
- Power of Three
- Power of Four

Why group them?
These problems share the mathematical "power" pattern, but differ in their optimal solutions. For two and four, bit tricks are possible, while three requires looping or recursion. Practicing all three cements your understanding and exposes you to multiple solution patterns.

Problem 1: Power of Two

Problem Statement (rephrased):

Given an integer n, determine if it is a power of 2. Return True if it is, otherwise False.

LeetCode link

Example:

Input: n = 16

Output: True

Because 16 = 2^4.

How to approach:

Think about what it means to be a power of two. If you keep dividing an even number by 2, you should eventually reach 1. If you hit an odd number (other than 1) or a non-positive number, it's not a power of two.

Try this test case on paper:

n = 18

Can you divide by 2 all the way to 1, without leftovers?

Brute-force approach:

Keep dividing n by 2 as long as it's even. If you reach 1, return True. If you hit an odd number (other than 1) or go negative, return False.

- **Time complexity:** O(log n) (since we halve n each time)

But there's a trick! For powers of two, only one bit is set in the binary representation. For example, 8 in binary is 1000.

Optimal Approach: Bit Manipulation

If n is a power of two, its binary representation has a single 1.

Check:

- n > 0
- n & (n - 1) == 0

This works because subtracting 1 from a power of two flips all the bits after the lone 1, so AND-ing them gives 0.

Python Solution

```
def isPowerOfTwo(n):
    """
    Returns True if n is a power of two, else False.
    """
    # Check for positive numbers only
    if n <= 0:
        return False
    # Check if only one bit is set in n
```

```
    return (n & (n - 1)) == 0
```

- **Time complexity:** O(1) (bitwise operations are constant time)
- **Space complexity:** O(1)

Breakdown:

- `n <= 0` filters out non-positive numbers (powers of 2 are always positive).
- `n & (n-1)` is a classic bit trick: for n = 8 (1000), n-1 = 7 (0111). 1000 & 0111 = 0000.
- Returns True iff n is a power of two.

Trace with n = 16:

- 16 is positive.
- 16 in binary: 10000
- 16 - 1 = 15, binary: 01111
- 10000 & 01111 = 0, so return True.

Try this case yourself:

n = 12

What does `12 & 11` yield? Is it zero? What does that mean?

Take a moment to solve this one on your own before moving on!

Fun fact:

Did you know you could also solve this by counting the number of ones in the binary representation? Try implementing that as a challenge!

Problem 2: Power of Three

How is this different?

You can't use bit tricks as easily for base 3, because powers of three don't line up with binary bit patterns. But the logic of repeated division still works.

Brute-force approach:

Divide n by 3 as long as it's divisible by 3. If you get to 1, it's a power of 3.

- **Time complexity:** O(log n) base 3.

Optimal approach:

- While n is divisible by 3, divide by 3.
- If you end at 1, return True. Otherwise, False.

Step-by-step:

- If n <= 0, return False.
- While n % 3 == 0:
  - n = n / 3
- If n == 1, return True; else, False.

Pseudocode

```
function isPowerOfThree(n):
    if n <= 0:
        return False
    while n % 3 == 0:
        n = n / 3
    return n == 1
```

Example:

Input: n = 27

- 27 % 3 == 0: 27 / 3 = 9
- 9 % 3 == 0: 9 / 3 = 3
- 3 % 3 == 0: 3 / 3 = 1
- n == 1, so return True.

Try this case:

n = 45

What happens as you divide by 3? Do you end at 1?

Trace for n = 9:

- 9 % 3 == 0: 9 / 3 = 3
- 3 % 3 == 0: 3 / 3 = 1
- n == 1 -> True

Time complexity: O(log n) base 3

Space complexity: O(1)

Problem 3: Power of Four

What's new here?

Power of 4 is like power of 2, but only even powers: 1, 4, 16, 64, ...

Challenge:

All powers of 4 are powers of 2, but not all powers of 2 are powers of 4! (8 is 2^3, but not 4^k.)

Approaches:

- Brute-force: Keep dividing by 4 until you reach 1.
- Bit manipulation: Power of four numbers have only one bit set, and that bit is in an odd position (1-based).

Step-by-step (Bit approach):

- Check n > 0.
- Check n is a power of 2: (n & (n - 1)) == 0

- Check the single set bit is in an odd position:
  - 0x55555555 is a mask with 1s at all odd positions.
  - (n & 0x55555555) != 0

Pseudocode

```
function isPowerOfFour(n):
    if n <= 0:
        return False
    if n & (n - 1) != 0:
        return False
    if n & 0x55555555 == 0:
        return False
    return True
```

Example:

n = 16

- 16 > 0
- 16 & (15) == 0 (so it's a power of 2)
- 16 & 0x55555555 == 16 (the 1-bit is in the correct place)
- Return True

Try this:

n = 8

Does it pass all the checks?

Trace for n = 64:

- 64 > 0
- 64 & 63 == 0
- 64 & 0x55555555 == 0 (64 is 2^6, but not 4^k; its bit is in an even position)
- Return False

Time complexity: O(1)

Space complexity: O(1)

Summary and Next Steps

Today, you explored the "power of X" pattern, learning how to check if a number is a power of 2, 3, or 4. You saw:

- **Division pattern:** Keep dividing by the base and check if you reach 1.
- **Bit tricks:** For powers of 2 (and 4), you can use bitwise operations for blazing-fast checks.
- **Masking:** For power of four, a mask ensures the set bit is in the correct position.

Common traps:

- Forgetting to check for non-positive numbers.

- Using bit tricks for bases other than 2 (doesn’t work!).
- Not handling large numbers or edge cases like n = 0 or negative numbers.

Action List

- Solve all three problems on your own, even the one with code above.
- For Power of Three and Four, try both division and alternative approaches.
- Challenge: Implement Power of Two using binary string counting.
- Find other “power of X” problems — what about 5 or 10?
- Compare your solutions to others and note how they handle edge cases.
- Remember: Getting stuck is part of the process. Keep practicing!

Happy coding!