Topic Introduction

Welcome to today's PrepLetter! We're diving into a powerful set of interview problems that blend **mathematical computation with clever algorithmic patterns**. Our focus: using **binary search** and **bit manipulation** to achieve blazing-fast calculations, even when brute force would be far too slow.

What are Binary Search and Bit Manipulation?

Binary search is a classic algorithm used to efficiently find a target value within a sorted sequence. Instead of checking each element one by one, it repeatedly divides the search space in half, zeroing in on the answer in logarithmic time. It's like guessing a number between 1 and 100, and each time someone tells you "higher" or "lower," you cut the range in half.

Bit manipulation refers to directly operating on the binary representations of numbers. This can unlock powerful tricks — like quickly multiplying or dividing by powers of two, or efficiently representing sets and flags.

Where are they useful?

- When calculations must be done without using built-in operators.
- When problems involve powers, roots, or division.
- Whenever brute-force is too slow for large numbers.

Simple Example (not one of our problems):

Suppose you want to find the smallest positive integer `n` such that `n^2` is greater than or equal to 40. Instead of checking `1, 2, 3...`, you could use binary search on the range [1, 40], check the square at midpoints, and quickly zoom in on the answer.

Today's trio of problems — **Divide Two Integers**, **Pow(x, n)**, and **Sqrt(x)** — all require you to compute mathematical results efficiently using binary search or bit tricks.

Why these three together? Each asks you to perform what seems like a simple math calculation (division, exponentiation, square root), but **without using the direct operator**. The optimal solutions all use binary search or bit shifts to achieve logarithmic efficiency.

Let's explore these one by one, starting with the most conceptually challenging: **Divide Two Integers**.

Problem 1: Divide Two Integers

Problem Statement (in plain English):
[Divide Two Integers - LeetCode]

Given two integers, `dividend` and `divisor`, compute the integer quotient after dividing `dividend` by `divisor` — **without using multiplication, division, or modulus operators**. The result should be truncated toward zero (like integer division in most languages), and you must handle overflow (if the result is outside the 32-bit signed integer range).

Example:
Input: `dividend = 10`, `divisor = 3`

Output: `3`

(10 divided by 3 is 3.333..., so the integer part is 3.)

Conceptual Walkthrough:

• At first glance, you might want to subtract the divisor from the dividend repeatedly until you can't anymore. But that would take O(N) time for large numbers!

• Instead, we can use **bit manipulation** to speed this up. Think of division like repeated subtraction, but instead of subtracting one divisor at a time, we subtract the largest possible multiple using bit shifts.

Try This One:

Input: `dividend = 43`, `divisor = 8`

What should the output be?

Brute-Force Approach:

• Repeatedly subtract `divisor` from `dividend` until what remains is less than `divisor`.

• This is O(N) where N is the absolute value of `dividend`.

Optimal Approach (Bit Manipulation):

• Use bit shifts to subtract large multiples of `divisor` at once.

  • For example, 8<<1 = 16, 8<<2 = 32, etc.

• At each step, subtract the highest shifted divisor that fits into the remaining dividend.

• Accumulate the multiples in the result.

• Handle negatives and overflow.

Let's see this in action!

```
def divide(dividend: int, divisor: int) -> int:
    # Special case: overflow
    INT_MAX = 2**31 - 1
    INT_MIN = -2**31
    if dividend == INT_MIN and divisor == -1:
        return INT_MAX

    # Work with negatives to avoid overflow
    negatives = 2
    if dividend > 0:
        dividend = -dividend
        negatives -= 1
    if divisor > 0:
        divisor = -divisor
        negatives -= 1

    quotient = 0
    # Subtract divisor multiples from dividend
    while dividend <= divisor:
        temp_divisor = divisor
```

```
        multiple = 1
        # Double until too big
        while dividend <= (temp_divisor << 1) and (temp_divisor << 1) < 0:
            temp_divisor <<= 1
            multiple <<= 1
        dividend -= temp_divisor
        quotient += multiple

    if negatives == 1:
        return -quotient
    else:
        return quotient
```

Time Complexity: O(log N), where N is the absolute value of the dividend

Space Complexity: O(1)

Explanation:

- Convert both numbers to negatives (avoids overflow from -2^31).
- For each step, find the largest shifted divisor that fits into the current dividend.
- Subtract that and record how many times it fit (by powers of 2).
- At the end, correct the sign.

Trace Example (dividend = 43, divisor = 8):

- Both numbers negative: dividend = -43, divisor = -8
- Largest multiple: -8 << 2 = -32 (fits), so subtract -32, quotient += 4
- Remaining: -43 - (-32) = -11
- -8 << 0 = -8 (fits), subtract -8, quotient += 1
- Remaining: -11 - (-8) = -3 (less than divisor), done!
- Quotient = 5

Try This One (for practice):

Input: `dividend = -27`, `divisor = 4`

What should the output be?

Encouragement:

Take a moment to try implementing this or stepping through it with pen and paper before peeking at the code!

Problem 2: Pow(x, n)

Problem Statement (in plain English):

[Pow(x, n) - LeetCode]

Implement a function to compute `x` raised to the power `n` (i.e., `x^n`). You must do this efficiently (in logarithmic time) and handle negative exponents.

Example:

Input: `x = 2.0`, `n = 10`

Output: `1024.0`

Why is this similar to the previous problem?

- Both use logarithmic patterns: we double (or halve) our work at each step.
- Here, we use **fast exponentiation** (exponentiation by squaring), which is analogous to using bit shifts to quickly accumulate results.

Brute-Force:

Multiply `x` by itself `n` times (O(n)).

Optimal Approach (Exponentiation by Squaring):

- If n == 0: return 1.
- If n < 0: invert x, use -n.
- If n is even: pow(x, n) = pow(x*x, n//2)
- If n is odd: pow(x, n) = x *pow(x*x, n//2)

Step-by-Step (pseudocode):

```
function myPow(x, n):
    if n == 0: return 1
    if n < 0:
        x = 1 / x
        n = -n
    result = 1
    while n > 0:
        if n % 2 == 1:
            result *= x
        x *= x
        n //= 2
    return result
```

Example Trace (x=2, n=10):

- n even: result *= x? No, n=10 is even.
- x = 2*2 = 4, n = 5
- n odd: result = *x -> result = 14* = 4
- x = 4*4 = 16, n = 2
- n even: x = 16*16=256, n=1
- n odd: result = 4*256=1024
- n=0, done!

Dry-run Case:

Try `x = 3.0`, `n = 5` — what should be the result?

Time Complexity: O(log n)

Space Complexity: O(1) (if iterative), O(log n) if recursive

Encouragement:

Try writing the iterative or recursive version on your own!

Problem 3: Sqrt(x)

Problem Statement (in plain English):

[Sqrt(x) - LeetCode]

Given a non-negative integer `x`, compute and return the integer part of its square root (i.e., the largest integer `k` such that `k*k <= x`). Do not use built-in sqrt functions.

What's tricky here?

- This is a classic use of **binary search** on the answer space.
- Need to avoid integer overflow when checking `mid*mid`.

Brute-Force:

Try every integer from 1 up to x, check if `i*i <= x`.

Optimal Approach (Binary Search):

- Set `left = 0`, `right = x`
- While left <= right:
    - mid = (left + right) // 2
    - if mid*mid == x: return mid
    - if mid*mid < x: left = mid+1
    - else: right = mid-1
- Return right (the integer part of sqrt)

Pseudocode:

```
function mySqrt(x):
    if x < 2: return x
    left = 1, right = x // 2
    while left <= right:
        mid = (left + right) // 2
        if mid*mid == x:
            return mid
        elif mid*mid < x:
            left = mid + 1
        else:
            right = mid - 1
    return right
```

Example Trace (x = 17):

- left=1, right=8
- mid=4, 4*4=16 < 17, left=5
- mid=6, 6*6=36 > 17, right=5
- mid=5, 5*5=25 > 17, right=4

- left=5, right=4 (stop), return 4

Dry-run Case:

Try `x = 26`. What should be returned?

Time Complexity: O(log x)
Space Complexity: O(1)

Encouragement:

Write the function and test it with the above and a few more cases. Reflect on how binary search helps you avoid unnecessary work.

Summary and Next Steps

Today you tackled three classic problems that all **replace brute force with logarithmic-speed algorithms** using binary search or bit manipulation:

- **Divide Two Integers:** Used bit shifts to subtract large multiples, mimicking division.
- **Pow(x, n):** Used exponentiation by squaring — doubling the work at each step for fast results.
- **Sqrt(x):** Used binary search on the answer to quickly find the integer square root.

Key Patterns:

- Binary search is not just for searching arrays — it's for narrowing down any monotonic function.
- Bit manipulation lets you speed up repeated addition/subtraction.
- Exponentiation by squaring is a must-have trick for both integers and floats.

Common Mistakes:

- Forgetting to handle negative numbers or overflow cases.
- Not considering integer overflow in mid*mid.
- Not truncating (e.g., for square root, division).

Action List

- **Solve all three problems on your own** — even the one with code provided here.
- **Try implementing Problem 2 and 3 recursively** as well as iteratively.
- **Challenge yourself:** Can you solve Pow(x, n) using binary search on the answer space?
- **Explore more:** Find and practice other problems using these patterns, like searching in rotated arrays, finding the kth root, or bitwise manipulation tricks.
- **Compare your solutions:** Look at others' solutions for edge cases and style improvements.
- Most importantly, **don't get discouraged if you get stuck** — these patterns take practice, and every attempt builds your skill!

Happy coding — and may your algorithms always run fast!