# POW(x,n) | Binary Exponentiation | Leetcode

## Summary

The video presents a detailed explanation of implementing a function to calculate **x raised to the power n** $(x^n)$, covering both brute force and optimized approaches, including handling edge cases such as negative exponents and integer overflow. The content is framed as part of a placement preparation series.

---

## Core Concepts and Problem Statement

- The problem is to compute **power(x, n)** which returns $x^n$, where:
  - **x** can be any numeric type (integer, double).
  - **n** is an integer (positive, negative, or zero).
- Important to confirm with the interviewer if **n** can be negative.
- The function should handle various edge cases, particularly when **n** is negative or at the extreme limits of the integer range.

---

## Brute Force Approach

- Initialize **answer** as a double with value 1.
- Loop from 1 to **n** (if **n** is positive), multiplying **answer** by **x** in each iteration.
- If **n** is negative:
  - Calculate $x^{|n|}$ using the same loop.
  - Then return $\frac{1}{x^{|n|}}$.
- Edge case:
  - The absolute value of the minimum integer (e.g., -2147483648) cannot be stored in a positive integer variable due to overflow.
  - Use a larger data type such as `long` or `long long` to handle this safely.

- Time complexity: **O(n)**.
- Space complexity: **O(1)**.

---

## Optimized Approach: Binary Exponentiation (Exponentiation by Squaring)

- Key insight:

  For even powers:

  $$x^n = (x \times x)^{n/2}$$

  For odd powers:

  $$x^n = x \times x^{n-1}$$

- Algorithm steps:
  - If **n** is even:
    - Square **x** (i.e., **x ⇒ x * x**).
    - Halve **n** (i.e., **n ⇒ n / 2**).
  - If **n** is odd:
    - Multiply **answer** by **x**.
    - Decrement **n** by 1 to make it even.
  - Stop when **n** reduces to zero.
- When **n** is negative, convert it to positive (using a `long` to avoid overflow), apply the binary exponentiation, then take the reciprocal (1/answer).
- Time complexity: **O(log n)**.
- Space complexity: **O(1)**.

---

## Example Walkthrough: Calculating $2^{10}$

| Step | Operation | Explanation | Result |
|---|---|---|---|
| 1 | $2^{10}$ | $2^{2\times5} = (2^2)^5$ | $4^5$ |
| 2 | $4^5$ | $4 \times 4^4$ | $4 \times 256$ |
| 3 | $4^4$ | $(4^2)^2 = 16^2$ | 256 |
| 4 | $16^2$ | $16 \times 16^1$ | $16 \times 16$ |
| 5 | $16^1$ | $16 \times 16^0$ | $16 \times 1$ |
| 6 | $16^0$ | Base case: 1 | 1 |
| 7 | Calculated $4^4 = 256$ | Multiply to get $4^5 = 4 * 256 = 1024$ | 1024 |
| 8 | Final result: $2^{10} = 1024$ | | 1024 |

## Code Outline

- Initialize `answer = 1.0`.
- Store a copy of `n` in a larger integer type (`long`) to prevent overflow on negation.
- If `n` is negative, convert `n` to positive.
- Loop while `n > 0`:
  - If `n` is odd, multiply `answer` by `x` and decrement `n` by 1.
  - If `n` is even, square `x` and halve `n`.
- After loop ends:
  - If the original `n` was negative, return `1 / answer`.
  - Otherwise, return `answer`.

## Key Insights

- **Handling negative exponents** requires taking the reciprocal after computing the positive power.
- **Integer overflow with negative minimum value** must be managed by using a wider data type (`long` or `long long`).

- **Binary exponentiation optimizes the power calculation** from O(n) to O(log n) time.
- The approach works for both integer and floating-point bases.
- The base case of any number to the power 0 is always 1.

## Summary of Complexity

| Approach | Time Complexity | Space Complexity | Notes |
|---|---|---|---|
| Brute force | O(n) | O(1) | Simple loop, inefficient for large n |
| Binary Exponentiation | O(log n) | O(1) | Efficient, uses divide and conquer |

## Final Notes

- The video encourages viewers to like and share if they find the explanation helpful.
- Links to additional social platforms and groups are provided for further engagement.
- Future videos will continue exploring problems from the "sd sheet" for placement preparation.

**This explanation ensures clarity on the power function implementation, covers edge cases, and provides optimal solutions suitable for technical interviews.**

## 50. Pow(x, n)

Medium 1685 3200 Add to List Share

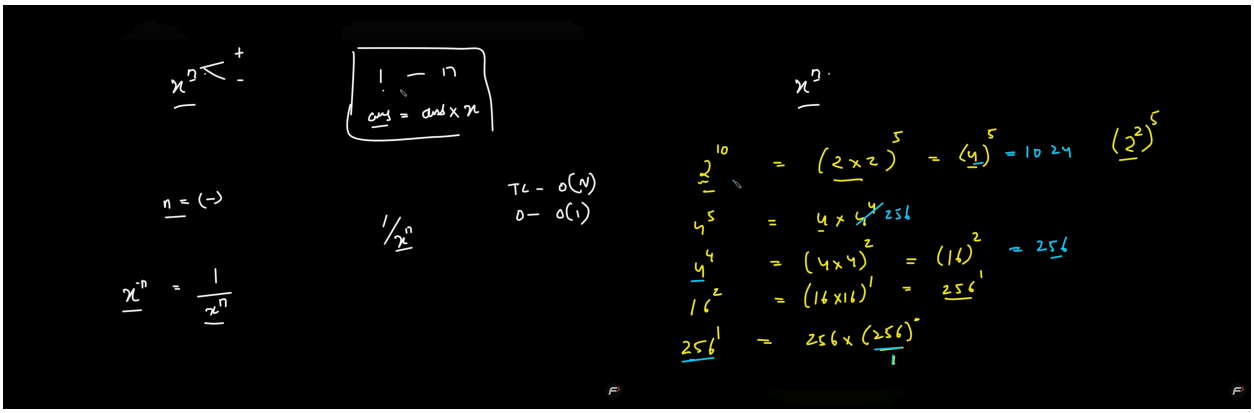
Implement pow(*x*, *n*), which calculates *x* raised to the power *n* (i.e. $x^n$).

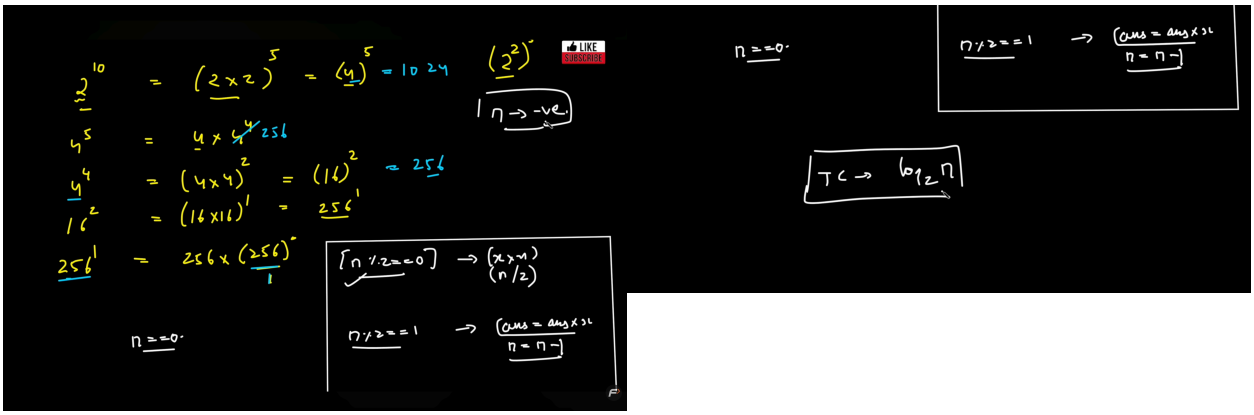
**Example 1:**

```
Input: x = 2.00000, n = 10
Output: 1024.00000
```

**Example 2:**

```
Input: x = 2.10000, n = 3
Output: 9.26100
```



## Code

### *Recursive Solution*

```
class Solution:
    def rec(self,x,n):
        if n==0:
            return 1
        p=self.rec(x,n//2)
        return x*p*p if n%2 else p*p
    def myPow(self,x:float,n:int)→float:
        ans=self.rec(x,abs(n))
        if n>0:
            return ans
        return 1/ans
```

### *Iterative Solution*

```
class Solution:
    def iter(self,x,n):
        ans=1
        N=n
        n=abs(n)
        while n:
            if n%2:
                ans=ans*x
                n=n-1
            else:
                x=x*x
                n=n//2
        return ans
    def myPow(self,x:float,n:int)→float:
        ans=self.iter(x,abs(n))
        if n>0:
            return ans
        return 1/ans
```