# **Leetcode 50 – Pow(x, n)**

## Problem Understanding

Implement pow(x, n) which calculates and returns xⁿ.

### Constraints:

* x can be any real number.
* n is an integer (can be negative).
* You must **optimize** for **logarithmic time**.

### Examples:

Input: x = 2.0, n = 10 → Output: 1024.0

Input: x = 2.1, n = 3 → Output: 9.261

Input: x = 2.0, n = -2 → Output: 0.25

## Optimized Java Solution (Exponentiation by Squaring)

class Solution {

public double myPow(double x, int n) {

long N = n; // avoid overflow for Integer.MIN\_VALUE

if (N < 0) {

x = 1 / x;

N = -N;

}

double result = 1.0;

while (N > 0) {

if ((N % 2) == 1) {

result \*= x;

}

x \*= x;

N /= 2;

}

return result;

}

}

## Dry Run Using Table

### Input:

x = 2.0, n = 10

|  |  |  |  |
| --- | --- | --- | --- |
| Step | N | x (base) | Result |
| 0 | 10 | 2.0 | 1.0 |
| 1 | 10→5 | 4.0 | 1.0 |
| 2 | 5→2 | 16.0 | 4.0 |
| 3 | 2→1 | 256.0 | 4.0 |
| 4 | 1→0 | 65536.0 | 1024.0 |

✅ Final Output: **1024.0**

## Time / Space Complexity

| **Metric** | **Value** |
| --- | --- |
| Time | O(log n) |
| Space | O(1) |

* Each time we **divide n by 2** → logarithmic steps
* Space is constant (iterative)

## Alternate Approaches

|  |  |  |  |
| --- | --- | --- | --- |
| Approach | Time | Space | Notes |
| ✅ Binary Exponentiation | O(log n) | O(1) | Best, iterative version |
| Recursive Binary Exponent | O(log n) | O(log n) | Easy to code but uses stack |
| Brute Force (multiply n times) | O(n) | O(1) | Not acceptable for large n |