50. Pow(x, n)

Medium Recursion Math

Problem Description

The task here is to implement a function that calculates the result of raising a number `x` to the power of `n`. In mathematical terms, you're asked to compute `x^n`. This function should work with `x` as a float (meaning it can be a decimal number) and `n` as an integer (which can be positive, negative, or zero).

Intuition

The intuitive approach to solve exponentiation could be to multiply `x` by itself `n` times. However, this method is not efficient for large values of `n` as it would have linear complexity O(n). Instead, we can use the "fast power algorithm," or "binary exponentiation," which reduces the time complexity substantially to O(log n).

This optimized solution is based on the principle that for any number `a` and integer `n`, the power `a^n` can be broken down into smaller powers using the properties of exponents. Specifically, if `n` is even, `a^n` can be expressed as `(a^(n/2))^2`, and if `n` is odd, it can be written as `a*(a^((n-1)/2))^2`. Applying these properties repeatedly allows us to reduce the problem into smaller subproblems, efficiently using divide-and-conquer strategy.

In this code, the function `qpow` implements this efficient algorithm using a loop, squaring `a` and halving `n` at each iteration, and multiplying to the answer when `n` is odd (detected by `n & 1` which checks the least significant bit). The loop continues until `n` becomes 0, which is when we have multiplied enough factors into `ans` to give us our final result.

If the exponent `n` is negative, we calculate `x^(-n)` and then take its reciprocal, as by mathematical rule, `x^(-n)` equals `1/(x^n)`. This is why, in the final return statement, the solution checks if `n` is non-negative and either returns `qpow(x, n)` or `1 / qpow(x, -n)` accordingly.

Solution Approach

The solution uses a helper function called `qpow` to implement a fast exponentiation algorithm iteratively. The function takes two parameters, `a` which represents the base number `x`, and `n` which represents the exponent. Here's a walkthrough of how the algorithm works:

1. **Initialize an accumulator (`ans`)**: A variable `ans` is initialized to 1. This will eventually hold the final result after accumulating the necessary multiplications.
2. **Iterative process**: A while loop is set up to run as long as `n` is not zero. Each iteration of this loop represents a step in the exponentiation process.
3. **Checking if `n` is odd (using bitwise AND)**: Within the loop, we check if the least significant bit of `n` is 1 by using `n & 1`. This is equivalent to checking if `n` is odd. If it is, we multiply the current `ans` by `a` because we know we need to include this factor in our product.
4. **Doubling the base (`a *= a`)**: After dealing with the possibility of `n` being odd, we square the base `a` by multiplying it by itself. This corresponds to the exponential property that `a^n = (a^(n/2))^2` for even `n`.
5. **Halving the exponent (using bitwise shift)**: We then halve the exponent `n` by right-shifting it one bit using `n >>= 1`. This is equivalent to integer division by 2. Squaring `a` and halving `n` is repeated until `n` becomes zero.
6. **Handling negative exponents**: After calculating the positive power using `qpow`, if the original exponent `n` was negative, we take the reciprocal of the result by returning `1 / qpow(x, -n)`. This handles cases where `x` should be raised to a negative power.

Data Structures: The solution does not use any complicated data structures; it relies on simple variables to hold intermediate results (`ans`, `a`).

Patterns: The iterative process exploits the divide-and-conquer paradigm, breaking the problem down into smaller subproblems of squaring the base and halving the exponent.

Algorithm Efficiency: Because of the binary exponentiation technique, the time complexity of the algorithm is O(log n), which is very efficient even for very large values of `n`.

Example Walkthrough

Let's walk through the iterative fast exponentiation technique using an example where we want to calculate `3^4`, which is 3 raised to the power of 4.

1. **Initialize accumulator (`ans`)**: Set `ans = 1`. This will hold our final result.
2. **Assign base to `a` and exponent to `n`**: We start with `a = 3` and `n = 4`.
3. **Begin iterative process**: Since `n` is not zero, enter the while loop.
4. **First iteration (n = 4)**:
   - **Check if `n` is odd**: `n & 1` is `0`, meaning `n` is even. Skip multiplying `ans` by `a`.
   - **Square the base (`a *= a`)**: `a` becomes `9`.
   - **Halve the exponent (`n >>= 1`)**: `n` becomes `2`.
5. **Second iteration (n = 2)**:
   - **Check if `n` is odd**: `n & 1` is `0`, still even. Skip multiplying `ans` by `a`.
   - **Square the base (`a *= a`)**: `a` becomes `81`.
   - **Halve the exponent (`n >>= 1`)**: `n` becomes `1`.
6. **Third iteration (n = 1)**:
   - **Check if `n` is odd**: `n & 1` is `1`, meaning `n` is odd. Multiply `ans` by `a`: `ans` becomes `81`.
   - **Square the base (`a *= a`)**: Although `a` becomes `6561` we stop because the next step would make `n` zero.
7. **Exponent 0 check**: The while loop exits because `n` is now zero.

Since we never had a negative power, we don't need to consider taking the reciprocal of `ans`, and the final answer for `3^4` is in `ans`, which is `81`.

In this example, you can see we only had to iterate 3 times to compute `3^4`, which is more efficient than multiplying `3` by itself `4` times. The optimized algorithm has a significant advantage as the values of `n` increase in magnitude.

Solution Implementation

Python

```python
class Solution:
    def myPow(self, x: float, n: int) -> float:
        # Inner function to perform the quick exponentiation.
        # This reduces the number of multiplications needed.
        def quick_power(base: float, exponent: int) -> float:
            result = 1.0
            # Continue multiplying the base until the exponent is zero
            while exponent:
                # If exponent is odd, multiply the result by the base
                if exponent & 1:
                    result *= base
                # Square the base (equivalent to base = pow(base, 2))
                base *= base
                # Right shift exponent by 1 (equivalent to dividing by 2)
                exponent >>= 1
            return result

        # If n is non-negative, call quick power with x and n directly.
        # Otherwise, calculate the reciprocal of the positive power.
        return quick_power(x, n) if n >= 0 else 1 / quick_power(x, -n)
```

Java

```java
class Solution {
    public double myPow(double x, int n) {
        // If power n is non-negative, calculate power using helper method
        if (n >= 0) {
            return quickPow(x, n);
        } else {
            // If power n is negative, calculate the inverse of the power
            return 1 / quickPow(x, -(long) n);
        }
    }

    private double quickPow(double base, long exponent) {
        double result = 1; // Initialize result to neutral element for multiplication

        // Loop through all bits of the exponent
        while (exponent > 0) {
            // If the current bit is set, multiply the result by the base
            if ((exponent & 1) == 1) {
                result *= base;
            }
            // Square the base for the next bit in the exponent
            base *= base;
            // Shift exponent to the right to process the next bit
            exponent >>= 1;
        }

        // Return the final result of base raised to the exponent
        return result;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to perform exponentiation
    double myPow(double x, int n) {
        // Inner function to calculate power using Quick Power Algorithm (also known as Binary Exponentiation)
        auto quickPow = [](double base, long long exponent) -> double {
            double result = 1.0; // Initialize the result to 1, as anything to the power of 0 is 1
            while (exponent > 0) { // Iterate until the exponent becomes 0
                if (exponent & 1) { // If the exponent is odd, multiply the current result by the base
                    result *= base;
                }
                base *= base; // Square the base
                exponent >>= 1; // Right shift exponent by 1 (divide the exponent by 2)
            }
            return result; // Return the final result of base raised to the exponent
        };

        // Check the sign of the exponent and call quickPow function
        // If 'n' is negative, we take the reciprocal of base raised to the power of absolute value of 'n'
        // We cast 'n' to long long to handle cases when n is INT_MIN, which flips to positive out of range if n is an int
        return n >= 0 ? quickPow(x, n) : 1.0 / quickPow(x, -(long long) n);
    }
};
```

TypeScript

```typescript
function myPow(x: number, n: number): number {
    // A helper function to calculate the power using quick exponentiation
    const quickPow = (base: number, exponent: number): number => {
        let result = 1; // Initialize result to 1

        // Loop until the exponent becomes zero
        while (exponent) {
            // If exponent is odd, multiply the result by the current base
            if (exponent & 1) {
                result *= base;
            }
            base *= base; // Square the base
            exponent >>>= 1; // Right shift exponent by 1 bit to divide it by 2
        }
        return result; // Return the calculated power
    };

    // If the exponent n is non-negative, just use quickPow() directly
    // If the exponent n is negative, take the reciprocal of the positive power
    return n >= 0 ? quickPow(x, n) : 1 / quickPow(x, -n);
}
```


```python
class Solution:
    def myPow(self, x: float, n: int) -> float:
        # Inner function to perform the quick exponentiation.
        # This reduces the number of multiplications needed.
        def quick_power(base: float, exponent: int) -> float:
            result = 1.0
            # Continue multiplying the base until the exponent is zero
            while exponent:
                # If exponent is odd, multiply the result by the base
                if exponent & 1:
                    result *= base
                # Square the base (equivalent to base = pow(base, 2))
                base *= base
                # Right shift exponent by 1 (equivalent to dividing by 2)
                exponent >>= 1
            return result

        # If n is non-negative, call quick power with x and n directly.
        # Otherwise, calculate the reciprocal of the positive power.
        return quick_power(x, n) if n >= 0 else 1 / quick_power(x, -n)
```


Time and Space Complexity

The given code implements the binary exponentiation algorithm to calculate `x` raised to the power `n`.

Time Complexity

The time complexity of the algorithm is `O(log n)`. This is because the `while` loop runs for the number of bits in the binary representation of `n`. In each iteration, `n` is halved by the right shift operation (`n >>= 1`), which reduces the number of iterations to the logarithm of `n`, hence `O(log n)`.

Space Complexity

The space complexity of the algorithm is `O(1)`. The function `qpow` only uses a constant amount of additional space for variables `ans` and `a`, which are used to keep track of the cumulative product and the base that's being squared in each iteration. There is no use of any data structure that grows with the input size, so the space requirement remains constant irrespective of `n`.