50. Pow(x, n)

Dec. 27, 2017 | 94.9K views

Average Rating: 4.02 (50 votes)

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

Example 1:

```
Input: 2.00000, 10
Output: 1024.00000
```

Example 2:

```
Input: 2.10000, 3
Output: 9.26100
```

Example 3:

```
Input: 2.00000, -2
Output: 0.25000
Explanation: 2^-2 = 1/2^2 = 1/4 = 0.25
```

Note:

- -100.0 < *x* < 100.0
- *n* is a 32-bit signed integer, within the range $[-2^{31}, 2^{31} - 1]$

Approach 1: Brute Force

Intuition

Just simulate the process, multiply `x` for `n` times.

If $n < 0$, we can substitute x, n with $\dfrac{1}{x}, -n$ to make sure $n \ge 0$. This restriction can simplify our further discussion.

But we need to take care of the corner cases, especially different range limits for negative and positive integers.

Algorithm

We can use a straightforward loop to compute the result.

C++ Java Copy

```
class Solution {
public:
    double myPow(double x, int n) {
        long long N = n;
        if (N < 0) {
            x = 1 / x;
            N = -N;
        }
        double ans = 1;
        for (long long i = 0; i < N; i++)
            ans = ans * x;
        return ans;
    }
};
```

Complexity Analysis

- Time complexity : $O(n)$. We will multiply `x` for `n` times.
- Space complexity : $O(1)$. We only need one variable to store the final product of `x`.

Approach 2: Fast Power Algorithm Recursive

Intuition

Assuming we have got the result of x^n, how can we get x^{2*n} ? Obviously we do not need to multiply `x` for another `n` times. Using the formula $(x^n)^2 = x^{2*n}$, we can get x^{2*n} at the cost of only one computation. Using this optimization, we can reduce the time complexity of our algorithm.

Algorithm

Assume we have got the result of $x^{n/2}$, and now we want to get the result of x^n. Let `A` be result of $x^{n/2}$, we can talk about x^n based on the parity of `n` respectively. If `n` is even, we can use the formula $(x^n)^2 = x^{2*n}$ to get $x^n = A * A$. If `n` is odd, then $A * A = x^{n-1}$. Intuitively, We need to multiply another x to the result, so $x^n = A * A * x$. This approach can be easily implemented using recursion. We call this method "**Fast Power**", because we only need at most $O(\log n)$ computations to get x^n.

C++ Java Copy

```
class Solution {
public:
    double fastPow(double x, long long n) {
        if (n == 0) {
            return 1.0;
        }
        double half = fastPow(x, n / 2);
        if (n % 2 == 0) {
            return half * half;
        } else {
            return half * half * x;
        }
    }
    double myPow(double x, int n) {
        long long N = n;
        if (N < 0) {
            x = 1 / x;
            N = -N;
        }
        return fastPow(x, N);
    }
};
```

Complexity Analysis

- Time complexity : $O(\log n)$. Each time we apply the formula $(x^n)^2 = x^{2*n}$, n is reduced by half. Thus we need at most $O(\log n)$ computations to get the result.
- Space complexity : $O(\log n)$. For each computation, we need to store the result of $x^{n/2}$. We need to do the computation for $O(\log n)$ times, so the space complexity is $O(\log n)$.

Approach 3: Fast Power Algorithm Iterative

Intuition

Using the formula $x^{a+b} = x^a * x^b$, we can write `n` as a sum of positive integers, $n = \sum_i b_i$. If we can get the result of x^{b_i} quickly, the total time for computing x^n will be reduced.

Algorithm

We can use the binary representation of `n` to better understand the problem. Let the binary representation of `n` to be $b_1, b_2, ..., b_{length\_limit}$ from the Least Significant Bit(LSB) to the Most Significant Bit(MSB). For the `i` th bit, if $b_i = 1$, it means we need to multiply the result by x^{2^i}.

It seems to have no improvement with this representation, since $\sum_i b_i * 2^i = n$. But using the formula $(x^n)^2 = x^{2*n}$ we mentioned above, we can see some differences. Initially $x^1 = x$, and for each $i > 1$, we can use the result of $x^{2^{i-1}}$ to get x^{2^i} in one step. Since the number of b_i is at most $O(\log n)$, we can get all x^{2^i} in $O(\log n)$ time. After that, for all `i` s that satisfy $b_i = 1$, we can multiply x^{2^i} to the result. This also requires $O(\log n)$ time.

Using fast power recursively or iteratively are actually taking different paths towards the same goal. For more information about fast power algorithm, you can visit its wiki[1].

C++ Java Copy

```
class Solution {
public:
    double myPow(double x, int n) {
        long long N = n;
        if (N < 0) {
            x = 1 / x;
            N = -N;
        }
        double ans = 1;
        double current_product = x;
        for (long long i = N; i ; i /= 2) {
            if ((i % 2) == 1) {
                ans = ans * current_product;
            }
            current_product = current_product * current_product;
        }
        return ans;
    }
};
```

Complexity Analysis

- Time complexity : $O(\log n)$. For each bit of `n` 's binary representation, we will at most multiply once. So the total time complexity is $O(\log n)$.
- Space complexity : $O(1)$. We only need two variables for the current product and the final result of `x`.

Footnotes

1. https://en.wikipedia.org/wiki/Exponentiation_by_squaring

Rate this article:

Previous Next

‹ 1 2 ›