# Question

Implement [pow(*x*, *n*)](http://www.cplusplus.com/reference/valarray/pow/), which calculates *x* raised to the power *n* (i.e. xn).

**Example 1:**

**Input:** x = 2.00000, n = 10

**Output:** 1024.00000

**Example 2:**

**Input:** x = 2.10000, n = 3

**Output:** 9.26100

**Example 3:**

**Input:** x = 2.00000, n = -2

**Output:** 0.25000

**Explanation:** 2-2 = 1/22 = 1/4 = 0.25

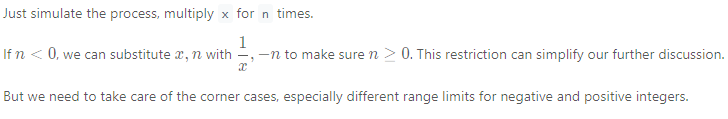
**Constraints:**

* -100.0 < x < 100.0
* -231 <= n <= 231-1
* -104 <= xn <= 104

# Solution

#### **Approach 1: Brute Force**

**Intuition**



**Algorithm**

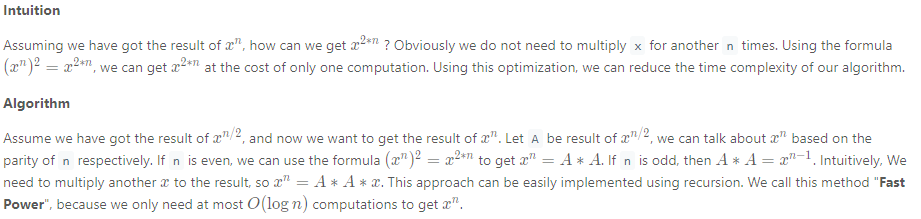
We can use a straightforward loop to compute the result.

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

**Complexity Analysis**

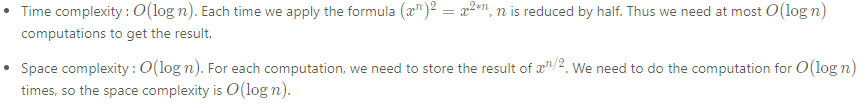
* Time complexity : O(n)*O*(*n*). We will multiply x for n times.
* Space complexity : O(1)*O*(1). We only need one variable to store the final product of x.

#### **Approach 2: Fast Power Algorithm Recursive**

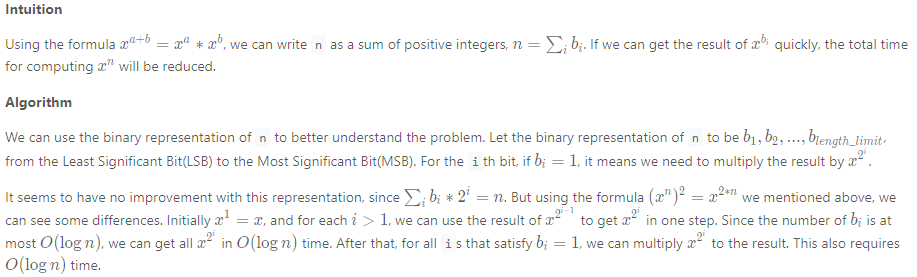


|  |
| --- |
| class Solution {  private double fastPow(double x, long n) {  if (n == 0) {  return 1.0;  }  double half = fastPow(x, n / 2);  if (n % 2 == 0) {  return half \* half;  } else {  return half \* half \* x;  }  }  public double myPow(double x, int n) {  long N = n;  if (N < 0) {  x = 1 /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

**Complexity Analysis**



#### **Approach 3: Fast Power Algorithm Iterative**



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]](https://leetcode.com/problems/powx-n/solution/#fn1).

|  |
| --- |
| class Solution {  public double myPow(double x, int n) {  long N = n;  if (N < 0) {  x = 1 / x;  N = -N;  }  double ans = 1;  double current\_product = x;  for (long i = N; i > 0; 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)*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)*O*(log*n*).
* Space complexity : O(1)*O*(1). We only need two variables for the current product and the final result of x.