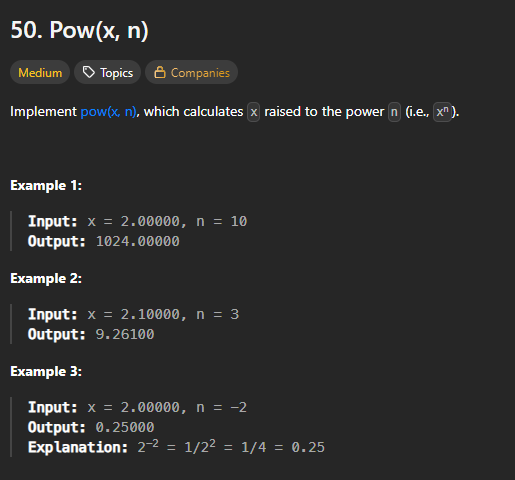
**LeetCode Problem :**

**Problem Statement**



**Code Snippet**

class Solution {

public:

double myPow(double x, int n) {

//Brute Force Approach

if(n==0) return 1.0; //base condition

double result = 1.0; //declaration cum initialisation

if(n>0){ //for power being positive

while(n!=0){

result \*= x;

n--; // counter update

}

}

else{ //for power being negative

while(n!=0){

result /= x;

n++;

}

}

return result;

}

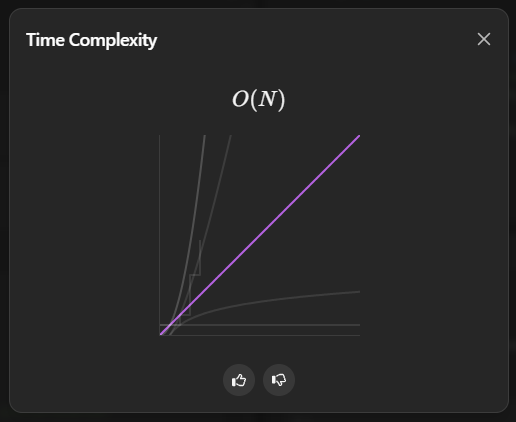
};

**Output**

Ran successfully | All Test Cases Passed

Error : Time Limit Exceeded (TLE)

Time & Space Complexity



**Optimised Code :**

class Solution {

public:

double myPow(double x, int n) {

// optimised approach Squaring Approach

long power = n; // to remove edge case error data overflow absolute (int min value 's) > (int max value)

if (power == 0)

return 1.0;

// a variable to store the value of result

double result = 1.0;

if (power< 0){

power = -power; //negative power to +ve

x = 1 /x; // by reciprocaling the base , power results will be accurate

}

while (power!= 0) {

// suppose x = 2, power= 20 then we can write 2^20 as (2 ^ 2) ^

// (20 / 2) (2 ^ 2 ) can be coded as x \* x and (20 / 2) can be

// coded as power/ 2

if (power% 2 == 0) {

x \*= x; // square base

power= power/ 2; // half the power

}

// if power is odd then we have to reduce power by one ,

// and multiple base by result to remove that extra multiple.

// suppose 2^21 can be written as 2 \* (2 ^ 20).

// power powerwill be reduced by 1 ,

// and removal of base will be multiplying it to result thus

// securing value

else {

result \*= x;

power--;

}

}

// } else {

// // convert powerinto +ve number

// while (power!= 0) {

// // suppose x = 2, power= -20 then we can write (1/2)^20 as (1/(2 ^

// // 2)) ^ (20 / 2) (2 ^ 2 ) can be coded as x \* x and (20 / 2)

// // can be coded as power/ 2

// if (power% 2 == 0) {

// x \*= x; // square base

// power= power/ 2; // half the power

// }

// // if power is odd then we have to reduce power by one ,

// // and divide result by base to remove that extra multiple.

// // suppose 2^21 can be written as 2 \* (2 ^ 20).

// // power powerwill be reduced by 1 ,

// // and removal of base will be dividing it to result thus

// // securing value

// else {

// result /= x;

// }

// }

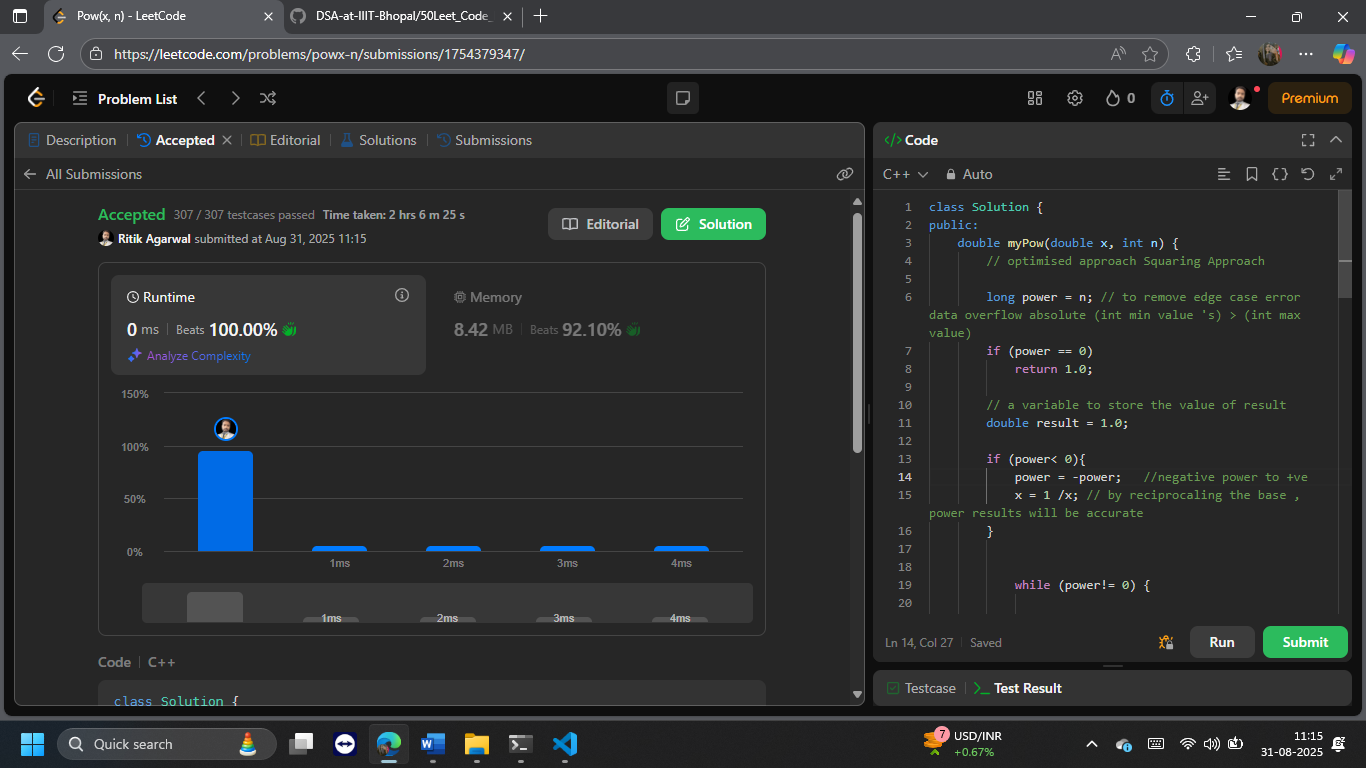
// }

return result;

}

};

**Optimised Code Result: O(log n)**



**My Learnings**

1. any code that takes more than 10^8 operations won’t be accepted.
2. **❌ Problem 2: power = -n causes overflow when n == Integer.MIN\_VALUE**

— especially for edge cases like n = Integer.MIN\_VALUE.

**🔍 Why?**

In C++, int ranges from -2,147,483,648 to 2,147,483,647. So:

int n = INT\_MIN; // -2147483648

int power = -n; // This overflows! Still equals INT\_MIN

Negating INT\_MIN exceeds the positive range of int, causing **undefined behavior**. Even though you declared power as long, you're still assigning -n while n is an int, so the overflow happens **before** the assignment.

**✅ Fix (without changing your code):**

Instead of power = -n;, you should write:

cpp

power = -power;

This safely negates the already widened long value.