Dear reader, welcome to the combined articles on the problems named **‘**[**Power – Linear**](https://www.pepcoding.com/resources/online-java-foundation/introduction-to-recursion/power-linear-official/ojquestion)’, and **‘**[**Power – Logarithmic**](https://www.pepcoding.com/resources/online-java-foundation/introduction-to-recursion/power-logarithmic-official/ojquestion)’.

***Problem Statement:*** Given a number x, and another number n, you are required to calculate x raised to the power n i.e. x multiplied n times.

*Note*: If you have not tried enough to come up with logic, then we recommend you to first spend an hour or so doing it, else read only the logic used, take it as a hint and try the problem again with the same logic.

***(Power - Linear) Solution: O(n) Time & O(1) Space***

For any recursive solution, you must define the expectation of the recursive function, then define a smaller sub-problem as the faith of the function, and finally derive a recursive relation using the faith and expectation.

***Expectation***: You must first define the expectation of the recursive function . It will give you the product of x multiplied n times i.e.

***Faith:*** Now, you should bring in faith in the recursive function (representing a smaller sub-problem) that on calling it, you will get the product of x multiplied n-1 times i.e.

***Recursive Relation:*** Now you need to meet the expectation with faith. Rearranging the first function we get,

When you will compare with , then you will get the following recursive relation:

Expectation, faith, and recursive relation are explained by our team in his video ‘[Pepcoding – Linear](https://www.youtube.com/watch?v=EohIyWnQYBY)’ from timestamp [*0:22 ,1:32*].

Are we done with the recursion, by defining a recursive relation? No, we are yet to define the base of the recursive function.

***Base Case***: Now, you should think about what can be the base case*.* Mathematically, if , then irrespective of the value of x, the value of will be 1. Hence you can return 1 as soon as n becomes 0.

***Important Note***: If the base case is not handled, then the above recursive call will keep on calling for negative values of n, and the recursion call stack will overflow giving a run-time error.

***Pseudo Code for*** ***Function***

1. If n is 0, then return 1.
2. Else
   1. Get the value of
   2. Return that value multiplied by x

*Note*: Before reading the Code, we recommend that you must try to come up with the solution on your own.

Now, hoping that you have tried by yourself, here is the Java code.

**Java Code**

*// Java Code to Calculate x^n in Linear Time Complexity*

import java.io.\*;

import java.util.\*;

public class Main {

  public static void main(String[] args) throws Exception {

    BufferedReader br = *new*

BufferedReader(*new* InputStreamReader(System.in));

    int x = Integer.parseInt(br.readLine());

    int n = Integer.parseInt(br.readLine());

    int p = power(x, n);

    System.out.println(p);

  }

  public static int power(int x, int n) {

*if*(n == 0){

*return* 1;

    }

    int xpnm1 = power(x, n - 1);

    int xpn = xpnm1 \* x;

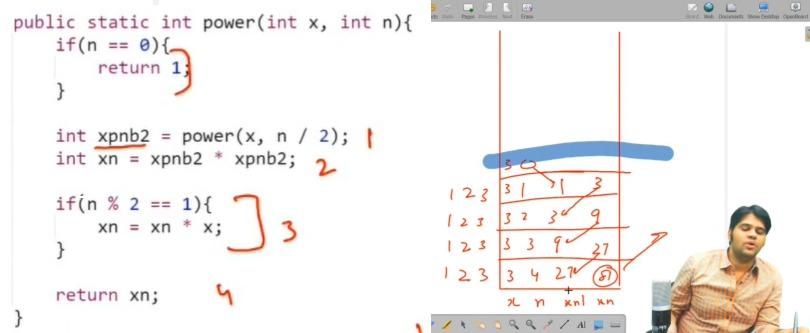
*return* xpn;

  }

}

Java Code is written and explained by our team in the same video from timestamp [*1:33 , 3:22]*.

If you are confused how the above recursive function works, then you should draw a ***recursion call stack*** and see how recursive calls are made.



The recursive call stack for a sample input is explained in the video from timestamp [*3:30* , *6:00]*.

* What is the ***time complexity*** of the above code?

Since you are recursively calling for a subproblem with n-1 from n, hence n + 1 recursive calls will be made (+ 1 when it hits the base case n = 0). You can also count the number of recursive calls in the call stack. So the time complexity turns out to be ***O(n)*** (which is independent of the value of x).

* What is the ***space complexity*** of the above code?

There is no data structure used, thus no auxiliary space is used. Hence, space complexity is ***O(1)***.

***Note***: There are n recursive calls that do take stack space, but we do not account for this space as we are only concerned with space used by data structures defined by us.

However, if it is mentioned to give the space complexity which takes into account the space taken by recursion call stack also, then, since there are n recursive calls, hence ***O(n) space*** will be required.

***Follow Up***:

* Don’t you feel that O(n) time for calculating just x^n is way too expensive?
* Can you improve the time complexity of the above problem?
* Can you derive a better recursive relation?

***Hint***: Try to think about the divide and conquer strategy. Try to divide the problem into fewer subproblems.

***(Power - Logarithmic) Solution: O(log2 n) Time & O(1) Space***

Before reading the algorithm used, please try this problem on your own using the hint provided.

Now, since you may have tried enough, here is the better solution to the problem stated above.

***Expectation***: Your expectation with the function should remain the same i.e.

***Faith***: You need to think of a more reduced sub-problem which you will have faith on. Can you break in terms of instead of .

Are you able to think of the formula mentioned below:

But is this formula valid for all positive values of n? What about odd positive integers?

Though the above formula holds good for positive even integral values of n, in the case of odd positive integers, your formula should look like

{ where n/2 is the floor value of the resultant (since n = odd number) }

So you can have faith in a recursive function , that it will give you the value of x when multiplied n/2 times i.e.

***Recursive Relation:*** Now, you need to meet the expectation with your faith.

If n is odd { n/2 is the floor value when n is divided by 2}

Else n is even

Expectation, faith, and the recursive relation are explained by our team in the video ‘[Power – Logarithmic](https://www.youtube.com/watch?v=O84uumjBOMY)’ from timestamp *0:18 to 3:32*.

***Base Case:*** As in the previous case, the base case will remain the same i.e. irrespective of the value of x. Hence, the base case will be when n becomes 0.

***Pseudo Code for*** ***Function:***

1. If n is 0, then return 1.
2. Else
3. Get the value of in a variable named ‘*xpnb2*’
4. If n is odd, then return
5. Else (n is even), return

*Suggestion*: You can give any name to your variables, but giving meaningful variable names, which can depict what value it is storing, where it will be used, etc. helps you debug your code easily and work on the code in the open-source community easily. Also, it gives a good impression about the candidate in an interview to the interviewer.

*Note*: Before reading the Code, we recommend that you must try to come up with the solution on your own. Now, hoping that you have tried by yourself, here is the Java code.

***Java Code***

import java.io.\*;

import java.util.\*;

public class Main {

  public static void main(String[] args) throws Exception {

    BufferedReader br = *new*

BufferedReader(*new* InputStreamReader(System.in));

    int x = Integer.parseInt(br.readLine());

    int n = Integer.parseInt(br.readLine());

    int p = power(x, n);

    System.out.println(p);

  }

  public static int power(int x, int n) {

*if*(n == 0){

*return* 1;

    }

    int xpnb2 = power(x, n / 2);

*// n/2 will automatically get floor value*

*// since it is type casted to int.*

    int xpn = xpnb2 \* xpnb2;

*if*(n % 2 == 1){

      xpn = xpn \* x;

    }

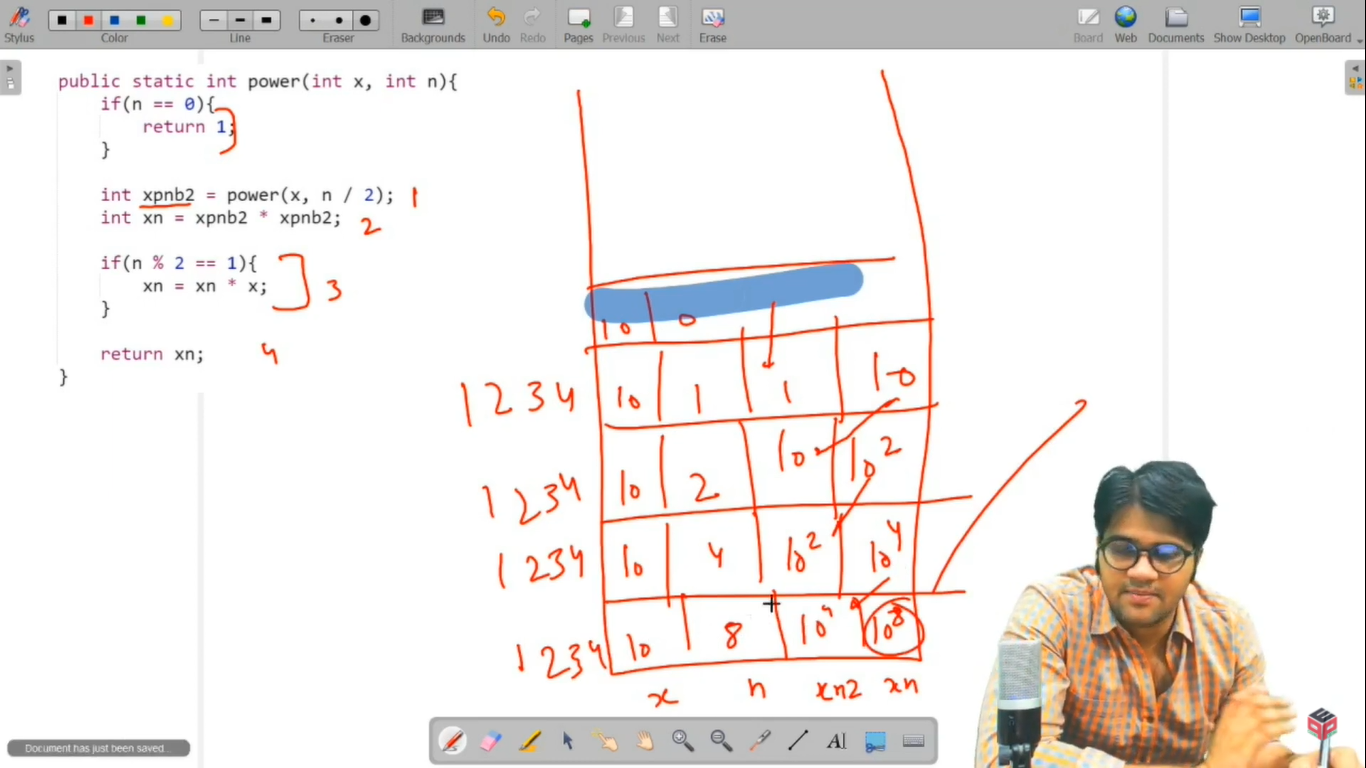
*return* xpn;

  }

}

Java code is written and explained by our team in the same video from timestamp [*3:33 to 5:42]*.

If you doubt how the above recursive function works, then you should draw a ***recursion call stack*** and see how recursive calls are made.



The recursive call stack for a sample input is explained in this part of the video solution [*5:42* to *8:58*].

* What is the ***time complexity*** of the above code?

Since you are recursively calling for a subproblem with n/2 from n, hence at max ***log2 (n)*** recursive calls will be made. You can also count the number of recursive calls in the call stack. So the time complexity turns out to be ***O(log2 n)*** (which is independent of the value of x).

Are you wondering how reducing n to n/2 brings down the number of recursive calls from O(n) to O(log n)? Can you think of a ***mathematical proof***?

**Hint**: Try finding the pattern in the values of n in recursive calls and find the count of recursive calls using this sequence.

***Proof of Logarithmic Time Complexity:***

When you will draw the recursion call stack, you will see that the value of n reduces by a factor of 2 always, i.e. it reduces in the following sequence:

. (Leaving the base case for better understanding).

Can you see that the above pattern forms a **geometric progression (G.P.)** ?

Indeed, it is a geometric progression with the first term ***a = 1*** and the constant ratio ***r = 1/2***.

Let the number of recursive calls be k (which is in turn equal to the number of terms in G.P.)

Then, you can apply the formula of the kth term of G.P. which will be:

On solving the above equation we get, , then taking log2 on both sides of the equation we get , i.e. k is **O(log2 n)**.

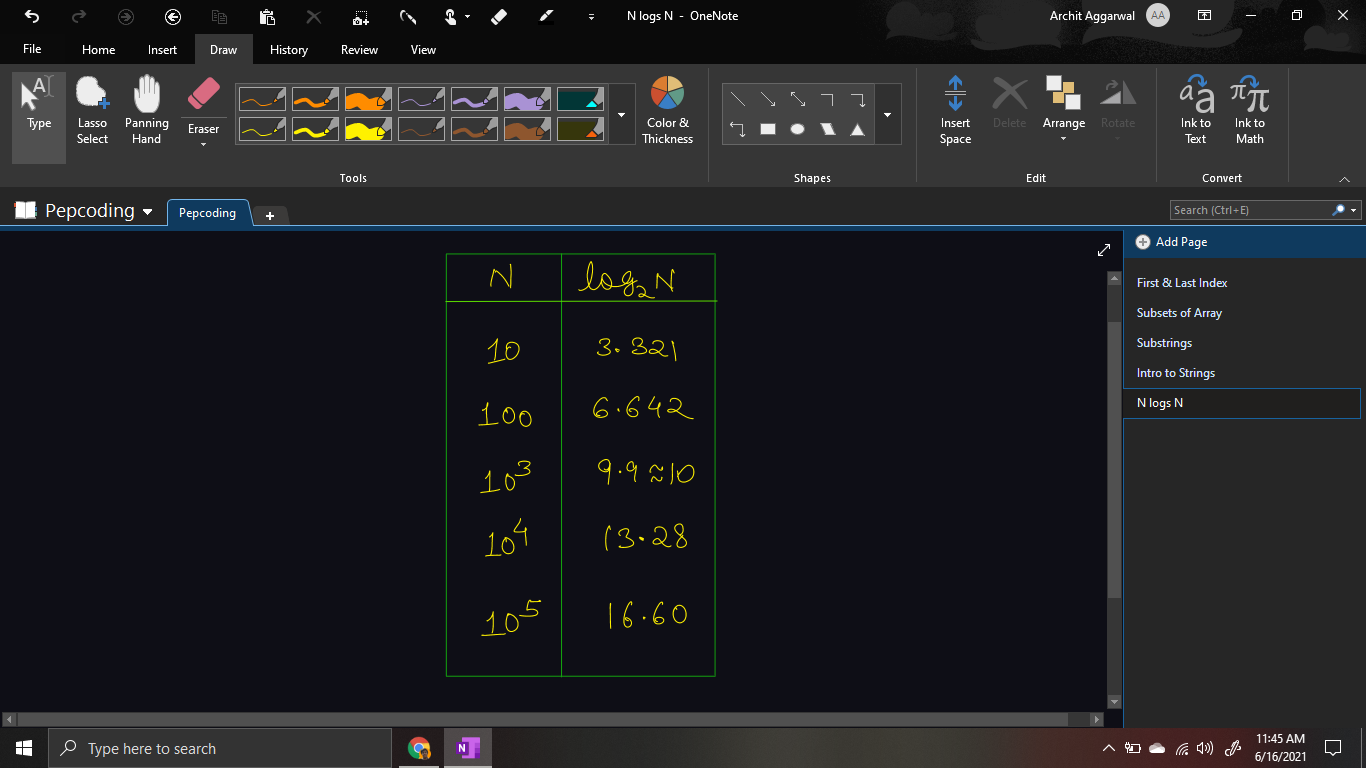
Since k is the number of recursive calls, hence the time complexity of the above solution is proved to be O(log2 n).

To follow this proof, you can go to the same video ‘Pepcoding – Logarithmic’, and watch the section: *9:12* to *12:53*.

* What is the ***difference*** in this logarithmic solution when compared to the linear complexity solution?

In the previous (linear complexity) solution, there were at max O(n) recursive calls made, hence O(n) time complexity, but here only at max O(log n) recursive calls are made, hence time complexity reduces to O(log n) only which is way less than O(n).

Calculate *log2 n* for n = 10, 100, 1000, 104, 105, etc. You will get to know that there is a significant difference between O(log2 n) and O(n).



This comparison is explained in the same video from *12:53* onwards using the sample input.

* What is the ***space complexity*** of the above code?

There is no data structure used, thus no auxiliary space is used. Hence, space complexity is ***O(1)***.

If the recursion call stack is taken into account, then space complexity will be O(log2 n) as there are k = (log2 n) recursive calls.

*Note*: Please remember that the above solution does not guarantee the right solution for large values of as such values cannot be stored in the ***int*** data type. Hence, trying the above solution might give unexpected results in such cases. Can you guess what we should do in this case?

There will be many methods which can help us solve for large values of n, and one of them can be to take modulo arithmetic. Don’t worry, we will learn about ***modular exponentiation*** in Level 3 - Number Theory & Mathematics Section.

*Note*: Don’t forget this problem, as you will face it again in the Bit Manipulation section where you will be asked to write an iterative solution to the same problem statement, by doing some manipulations in bits of n.

Hope that you liked the article on Power Function solved using 2 methods with linear time complexity and logarithmic time complexity.

Subscribe to Pepcoding’s youtube channel for more such amazing content on Data Structures & Algorithms and follow the resources available for all students in the resources section of Pepcoding’s website.

You can suggest any improvements to the article on our telegram channel, or on the youtube channel’s comment section.

Article Contributed by:

Archit Aggarwal