Nth Root of a Number using Binary Search

Problem Statement: Given two numbers N and M, find the Nth root of M. The nth root of a number M is defined as a number X when raised to the power N equals M. If the 'nth root is not an integer, return -1.

Example 1:

Input Format: N = 3, M = 27

Result: 3

Explanation: The cube root of 27 is equal to 3.

Example 2:

Input Format: N = 4, M = 69

Result: -1

Explanation: The 4th root of 69 does not exist. So, the answer is -1.

Solution:

Approach(Using linear search):

We can guarantee that our answer will lie between the range from 1 to m i.e. the given number. So, we will perform a linear search on this range and we will find the number x, such that
func(x, n) = m. If no such number exists, we will return -1.

Note: func(x, n) returns the value of x raised to the power n i.e. x^n.

Algorithm:

- We will first run a loop(***say i***) from 1 to m.

- **Inside the loop we will check the following:**
    - **If func(x, n) == m:** This means x is the number we are looking for. So, we will return x from this step.
    - **If func(x, n) > m:** This means we have got a bigger number than our answer and until now we have not found any number that can be our answer. In this case, our answer does not exist and we will break out from this step and return -1.

Note: We will use the power exponential method to implement the func() function and its time complexity will be O(logN)(*where N = given exponent*).

```
#include <bits/stdc++.h>
using namespace std;

// Power exponential method:
long long func(int b, int exp) {
   long long  ans = 1;
   long long base = b;
   while (exp > 0) {
      if (exp % 2) {
         exp--;
         ans = ans * base;
      }
      else {
         exp /= 2;
         base = base * base;
      }
   }
```

```
    return ans;
}

int NthRoot(int n, int m) {
    //Use linear search on the answer space:
    for (int i = 1; i <= m; i++) {
        long long val = func(i, n);
        if (val == m * 1ll) return i;
        else if (val > m * 1ll) break;
    }
    return -1;
}

int main()
{
    int n = 3, m = 27;
    int ans = NthRoot(n, m);
    cout << "The answer is: " << ans << "\n";
    return 0;
}
```

Time Complexity: O(M), M = the given number.
Reason: Since we are using linear search, we traverse the entire answer space.

Space Complexity: O(1) as we are not using any extra space.

Optimal Approach(Using Binary Search):

We are going to use the Binary Search algorithm to optimize the approach.

The primary objective of the Binary Search algorithm is to efficiently determine the appropriate half to eliminate, thereby reducing the search space by half. It does this by determining a specific condition that ensures that the target is not present in that half.

Now, we are not given any sorted array on which we can apply binary search. But, if we observe closely, we can notice that our answer space i.e. [1, n] is sorted. So, we will apply binary search on the answer space.

Edge case: How to eliminate the halves:

Our first approach should be the following:

- After placing low at 1 and high m, we will calculate the value of 'mid'.
- Now, based on the value of 'mid' raised to the power n, we will check if 'mid' can be our answer, and based on this value we will also eliminate the halves. If the value is smaller than m, we will eliminate the left half and if greater we will eliminate the right half.

But, if the given numbers m and n are big enough, we cannot store the value mid^n in a variable. So to resolve this problem, we will implement a function like the following:

func(n, m, mid):

- We will first declare a variable 'ans' to store the value mid^n.
- Now, we will run a loop for n times to multiply the 'mid' n times with 'ans'.
- Inside the loop, **if at any point 'ans' becomes greater than m, we will return 2**.
- Once the loop is completed, if the 'ans' is equ**al to m, we will return 1**.
- **If the value is smaller, we will return 0**.

Now, based on the output of the above function, we will check if 'mid' is our possible answer or we will eliminate the halves. Thus we can avoid the integer overflow case.

Algorithm:

1. **Place the 2 pointers i.e. low and high:** Initially, we will place the pointers. The pointer low will point to 1 and the high will point to m.
2. **Calculate the 'mid':** Now, inside a loop, we will calculate the value of 'mid' using the following formula:
   mid = (low+high) // 2 ('//' refers to integer division)
3. **Eliminate the halves accordingly:**
    1. **If func(n, m, mid) == 1:** On satisfying this condition, we can conclude that the number 'mid' is our answer. So, we will return to 'mid'.
    2. **If func(n, m, mid) == 0:** On satisfying this condition, we can conclude that the number 'mid' is smaller than our answer. So, we will eliminate the left half and consider the right half(i.e. low = mid+1).
    3. **If func(n, m, mid) == 2:** the value mid is larger than the number we want. This means the numbers greater than 'mid' will not be our answers and the right half of 'mid' consists of such numbers. So, we will eliminate the right half and consider the left half(i.e. high = mid-1).
4. Finally, if we are outside the loop, this means no answer exists. So, we will return -1.

The steps from 2-3 will be inside a loop and the loop will continue until low crosses high.

```
#include <bits/stdc++.h>

using namespace std;
```

```
//return 1, if == m:
//return 0, if < m:
//return 2, if > m:
int func(int mid, int n, int m) {
    long long ans = 1;
    for (int i = 1; i <= n; i++) {
        ans = ans * mid;
        if (ans > m) return 2;
    }
    if (ans == m) return 1;
    return 0;
}

int NthRoot(int n, int m) {
    //Use Binary search on the answer space:
    int low = 1, high = m;
    while (low <= high) {
        int mid = (low + high) / 2;
        int midN = func(mid, n, m);
        if (midN == 1) {
            return mid;
        }
        else if (midN == 0) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}
```

```
int main()
{
   int n = 3, m = 27;
   int ans = NthRoot(n, m);
   cout << "The answer is: " << ans << "\n";
   return 0;
}
```

Complexity Analysis

Time Complexity: O(logN), N = size of the given array.
Reason: We are basically using binary search to find the minimum.

Space Complexity: O(1)
Reason: We have not used any extra data structures, this makes space complexity, even in the worst case as O(1).