	Marwari University **Faculty of Technology** **Department of Information and Communication Technology**	
Subject: Design and Analysis of Algorithms (01CT0512)	**Aim:** Exponential Function with O(N) and O(logN).	
Experiment No: 03	**Date:**	**Enrollment No: 92301733054**

I. Exponential Function using Iterative (Naive) Approach :-

Theory: -

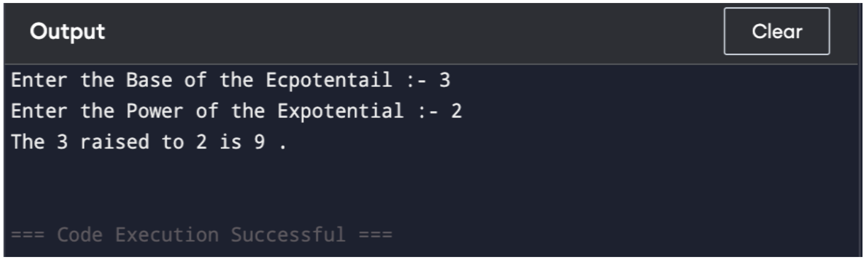
This method calculates the exponential value x^n by simply multiplying x by itself n times. It starts with a result of 1 and keeps multiplying it by x in each step until it has done this n times.While this approach is simple and easy to understand, it becomes inefficient for large values of n, since it performs one multiplication per step — resulting in a time complexity of O(n).

Programming Language: - C++

Code :-

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

long long Expotential(long base, long power) {
    if (power == 0) {
        return 1;
    }
    if (power == 1) {
        return base;
    }
    return base * Expotential(base, power - 1);
}

int main() {

    long base;
    long power;
    cout << "Enter the Base of the Ecpotentail :- ";
    cin >> base;
    cout << "Enter the Power of the Expotential :- ";
    cin >> power;

    long long result = Expotential(base, power);

    cout << "The " << base << " raised to " << power << " is " << result << " ." << endl;
    return 0;
}
```

Output :-



```
Enter the Base of the Ecpotentail :- 3
Enter the Power of the Expotential :- 2
The 3 raised to 2 is 9 .


=== Code Execution Successful ===
```

Space Complexity:- O(N)

Justification: -

The function uses recursion, and for each recursive call, a new stack frame is added to the call stack.So, if the power is n, the recursion will go n levels deep (from power down to 0).There is no additional data structure used, only the function call stack consumes space.

Time Complexity:O(1)

Best Case Time Complexity: O(1)

Justification: -

If power == 0, the function immediately returns 1.This is a constant-time operation and doesn't make any recursive call.

II. Exponential Function with O(N) using Divide and Conquer Approach :-

Theory: -This recursive method is smarter than the basic one because it uses a divide-and-conquer strategy to reduce the number of multiplications.

- If the exponent n is even, it calculates $x^{n/2}$ just once, and then squares the result.
- If n is odd, it calculates $x^{(n-1)/2}$, squares it, and then multiplies once more by x.

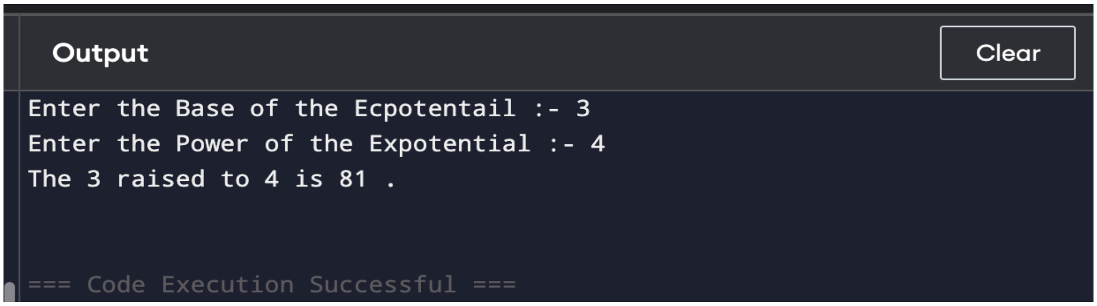
By breaking the problem in half each time, this approach significantly speeds things up, especially for large values of n.

Programming Language: - C++

Code :-

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

long long Expotential(long base, long power) {
  if (power == 0) {
    return 1;
    }

    else if (power % 2 == 0) {
      return Expotential(base, power / 2) * Expotential(base, power / 2);
    }
    else {
      return base * Expotential(base, power / 2) * Expotential(base, power / 2);
    }
}
int main() {

    long base;
    long power;
    cout << "Enter the Base of the Ecpotentail :- ";
    cin >> base;

    cout << "Enter the Power of the Expotential :- ";
    cin >> power;

    long long result = Expotential(base, power);
    cout << "The " << base << " raised to " << power << " is " << result << " ." << endl;
    return 0;
}
```

Output:-



```
Enter the Base of the Ecpotentail :- 3
Enter the Power of the Expotential :- 4
The 3 raised to 4 is 81 .


=== Code Execution Successful ===
```

Space Complexity:- O(power)
Justification: - Although the algorithm uses divide and conquer, it calls the recursive function twice in each step instead of storing the result and reusing it.This leads to redundant recursive calls, which results in a tree of recursive calls — similar to the naive recursion.Hence, the recursion depth can reach up to power in the worst case.So, space complexity remains O(power) due to the recursive call stack.

Time Complexity:-O(1)
Justification: - When power == 0, the function returns 1 immediately with no recursion, which takes constant time.

III. Exponential Function with O(logN) using Divide and Conquer Approach :-

Theory: -

This improved divide-and-conquer method uses the math behind exponentiation to make the process much faster:

- If the exponent nnn is even, it calculates $x^{n/2}$ once and squares it.
- If n is odd, it calculates $x^{(n-1)/2}$, squares that, and multiplies once more by x.

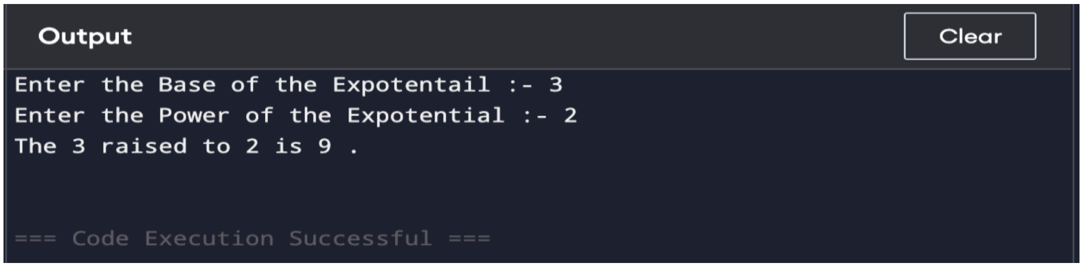
By cutting the exponent in half each time, this approach only needs about log(n) steps, making it very efficient — especially when dealing with large values of n. It significantly reduces the number of multiplications compared to the basic method.

Programming Language: - C++

Code :-

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

long long Expotential(long base, long power) {
   if (power == 0) {
      return 1;
   }
   if (base == 0) {
      return 0;
   }
   if (power < 0) {
      return  1 / Expotential(base, power * -1);
   }
   if (power == 1) {
      return base;
   }
   long long Half_Power = Expotential(base, power / 2);
   if (power % 2 == 0) {
      return Half_Power * Half_Power ;
   }
   else {
      return base * Half_Power * Half_Power ;
   }
}
int main() {
        long base;
        long power;
        cout << "Enter the Base of the Expotentail :- ";

        cin >> base;
        cout << "Enter the Power of the Expotential :- ";
        cin >> power;
        long long result = Expotential(base, power);
        cout << "The " << base << " raised to " << power << " is " << result << " ." << endl;
        return 0;
  }
```

Output:-



```
Enter the Base of the Expotentail :- 3
Enter the Power of the Expotential :- 2
The 3 raised to 2 is 9 .


=== Code Execution Successful ===
```

Space Complexity:-O(log power)
Justification: - The function uses recursion, and in each call, it divides the power by 2.This means the depth of the recursive call stack is $\log_2$(power) in the worst case.No extra memory (like arrays or maps) is used—just the call stack.

Time Complexity:O(1)
Justification: - When power == 0, the function immediately returns 1 without any recursion or multiplication. This takes constant time: O(1).

Conclusion:- In this experiment, we calculated x^n using three C++ methods—naive recursion, basic divide-and-conquer, and an optimized recursive approach—to understand their effect on time and space complexity.