Marwadi University Marwadi Chandarana Group NAAC A+	**Marwadi University** **Faculty of Engineering and Technology** **Department of Information and Communication Technology**	
Subject: DAA (01CT0512)	**AIM: Implementing the Exponential (Power) function to calculate x^n using Divide and Conquer Approach**	
Experiment No: 3	**Date:**	**Enrolment No: 92301733046**

Theory:

The exponential (power) function calculates x^n, where x is the base and n is the exponent. This operation can be done in different ways, each with different time and space requirements.

1. Iterative (Naive) Approach:

- This method multiplies the base x by itself n times in a loop. It is easy to implement but has a time complexity of O(n) in the worst case, as it performs n multiplications. The space complexity is O(1) because it uses only a fixed number of variables.

2. Divide and Conquer (Unoptimized, O(n)):

- This approach splits the problem into smaller subproblems by dividing n by 2 each time. However, it recalculates the same subproblem several times, leading to unnecessary work and a worst-case time complexity of O(n). The recursion depth lowers the space requirement to O(log n).

3. Divide and Conquer (Optimized, O(log n)):

- This version improves efficiency by calculating power(x, n//2) once and reusing the result. This dramatically reduces the number of multiplications, resulting in a worst-case time complexity of O(log n) and a space complexity of O(log n) due to recursion depth.

These three approaches show the balance between simplicity and efficiency in algorithm design. They illustrate how techniques like result reuse can greatly improve performance.

Programming Language: Python

Marwadi University Marwadi Chandarana Group NAAC A+	Marwadi University Faculty of Engineering and Technology Department of Information and Communication Technology	
Subject: DAA (01CT0512)	AIM: Implementing the Exponential (Power) function to calculate x^n using Divide and Conquer Approach	
Experiment No: 3	Date:	Enrolment No: 92301733046

1) **Exponential Function using Iterative (Naive) Approach**

Code:

```
def power_iterative(x, n):
    result = 1
    for i in range(n):
        result *= x
    return result


print(power_iterative(2, 3))
```

Output:

```
PS D:\DAA> python -u "d:\DAA\power_function.py"
8
PS D:\DAA>
```

Space complexity: O(1)

Justification: The algorithm uses only a fixed number of variables (result, x, n, and the loop counter), without any recursion or additional data structures. As a result, the space complexity remains constant at O(1) for all cases.

Time complexity:

Best case time complexity: O(1)

Justification: The best case occurs when n = 0. In this situation, the loop does not execute, and the function directly returns 1. This results in a constant-time operation with a time complexity of O(1).

Worst case time complexity: O(n)

Justification: The worst case occurs when n > 0. Here, the loop runs exactly n times, performing one multiplication per iteration. Therefore, the total number of operations grows linearly with n, giving a time complexity of O(n).

Marwadi University Marwadi Chandarana Group NAAC A+	**Marwadi University** **Faculty of Engineering and Technology** **Department of Information and Communication Technology**	
Subject: DAA (01CT0512)	**AIM: Implementing the Exponential (Power) function to calculate x^n using Divide and Conquer Approach**	
Experiment No: 3	**Date:**	**Enrolment No: 92301733046**

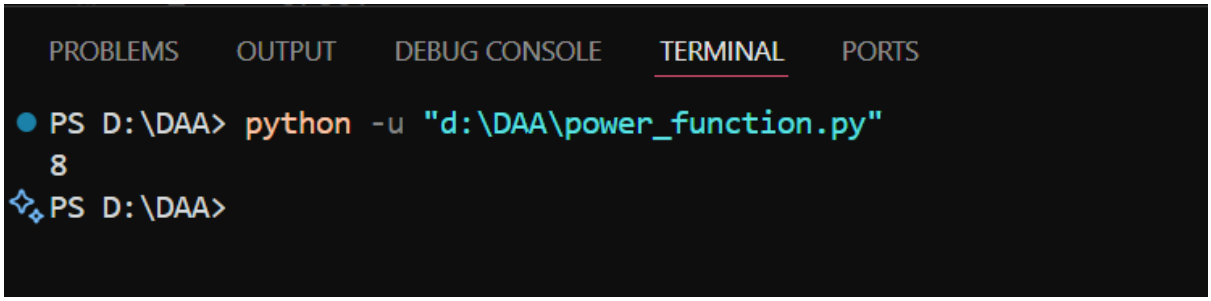
2) **Exponential Function with O(N) using Divide and Conquer Approach**

Code:

```
#t(n)=o(n)
def power(x,n):
    if n==0:
        return 1
    elif n%2==0:
        return power(x,n//2) * power(x,n//2)
    else:
        return x * power(x,n//2) * power(x,n//2)

print(power(2,3))
```

Output:



Space complexity: O(logn)

Justification: Each recursive call requires a separate location on the call stack when n > 0 in order to track variables and determine where to return once it is finished. It takes roughly $\log_2(n)$ steps to get to the base case because the function always reduces n by half. This indicates that the space complexity is O(log n), meaning that the call stack will never go deeper than roughly $\log_2(n)$ frames.

Time complexity:

Best case time complexity: O(1)

Justification: If it directly hit base case then it have O(1) time complexity

Worst case time complexity: O(n)

Justification: When n > 0, the function always makes two new recursive calls for the same half-sized problem, instead of saving and reusing the result. This means the work doubles at each step of the recursion. Even though the recursion only goes about $\log_2(n)$ levels deep, the total number of calls adds up quickly like 1 call, then 2 calls, then 4, then 8, and so on until it reaches roughly 2n calls. That's why, instead of being fast like O(log n), it ends up taking O(n) time.

Marwadi University Faculty of Engineering and Technology Department of Information and Communication Technology		
Subject: DAA (01CT0512)	AIM: Implementing the Exponential (Power) function to calculate x^n using Divide and Conquer Approach	
Experiment No: 3	Date:	Enrolment No: 92301733046

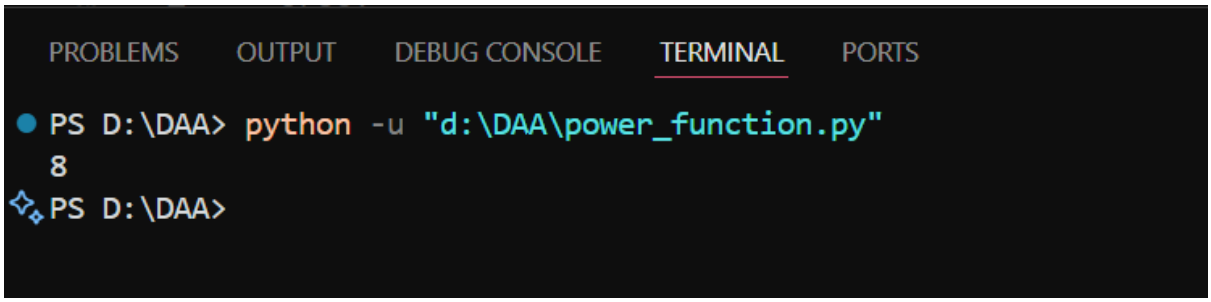
3) **Exponential Function with O(logN) using Divide and Conquer Approach**

Code:

```
#t(n)=o(log n)
# Using recursion with optimized approach
def power(x,n):
    temp=power(x,n//2)
    if n==0:
        return 1
    elif n%2==0:
        return temp * temp
    else:
        return x * temp * temp

print(power(2,3))
```

Output:



Marwadi University Marwadi Chandarana Group NAAC A+	**Marwadi University** **Faculty of Engineering and Technology** **Department of Information and Communication Technology**	
Subject: DAA (01CT0512)	**AIM: Implementing the Exponential (Power) function to calculate x^n using Divide and Conquer Approach**	
Experiment No: 3	**Date:**	**Enrolment No: 92301733046**

Space complexity: O(logn)

Justification: Each recursive call requires a separate location on the call stack when n > 0 in order to track variables and determine where to return once it is finished. It takes roughly $\log_2(n)$ steps to get to the base case because the function always reduces n by half. This indicates that the space complexity is O(log n), meaning that the call stack will never go deeper than roughly $\log_2(n)$ frames.

Time complexity:

Best case time complexity: O(1)

Justification: If it directly hit base case then it have O(1) time complexity

Worst case time complexity: O(logn)

Justification: In the worst scenario, the function calls power(x, n//2) exactly once in each step when n > 0. Regardless of how big n is, it does a tiny, fixed amount of additional work after returning the recursive result just a few multiplications. It only takes roughly $\log_2(n)$ steps to get to the base case, where n = 0, because each recursive step cuts n in half. This indicates that the function's worst-case time complexity is O(log n), meaning that the total amount of work increases proportionately to log n.