- **What is Dynamic Programming?**
  - An algorithmic technique for solving optimization problems.
  - Breaks down a problem into overlapping subproblems.
  - Solves each subproblem only once and stores the results (memoization).
  - Avoids recomputing the same subproblems repeatedly.
- **Key Characteristics:**
  - Optimal Substructure: An optimal solution to the problem contains optimal solutions to the subproblems.
  - Overlapping Subproblems: The subproblems are not independent; they share subsubproblems.
- **Two Main Approaches:**
  - Top-Down (Memoization): Start with the original problem and recursively break it down into subproblems. Store the results of each subproblem in a table (or memo) to avoid recomputation.
  - Bottom-Up (Tabulation): Start with the smallest subproblems and solve them first. Store the results in a table. Use the results of the smaller subproblems to solve the larger subproblems.

II. Fibonacci Numbers

- Definition: The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, usually starting with 0 and 1. So, the sequence is: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...
- Recursive Formula:
  - $F(0) = 0$
  - $F(1) = 1$
  - $F(n) = F(n-1) + F(n-2)$ for $n > 1$

III. Naive Recursive Implementation (Without Dynamic Programming)

- Show the straightforward recursive implementation of the Fibonacci sequence based on the recursive formula.
- Explain the inefficiency of this approach due to repeated calculations of the same Fibonacci numbers. Draw a recursion tree to illustrate the overlapping subproblems.

```
using System;
public class Fibonacci {
 public static int Fib(int n)
 {
 if (n <= 1)
 {
 return n;
 }
 else
 {
 return Fib(n - 1) + Fib(n - 2);
 }}
 public static void Main(string[] args)
 {
 int n = 6;
 Console.Write(Fib(n));
 Console.ReadKey();
 }
```

IV. Dynamic Programming Approaches for Fibonacci Numbers

- **A. Top-Down (Memoization)**
  1. Create a memo (e.g., an array or dictionary) to store the results of Fibonacci numbers. Initialize all entries to a special value (e.g., 0) to indicate that they haven't been computed yet.
  2. Implement the recursive function:
     - If the Fibonacci number for 'n' is already in the memo, return it.
     - Otherwise, compute F(n) recursively as F(n-1) + F(n-2).
     - Store the result in the memo before returning it.
  3. Demonstrate how memoization avoids redundant calculations.

```
using System;
```

```
class Program
{
    static int Fibonacci(int n, int[] values)
    {
        if (n <= 1) return 1; // الحالة الأساسية
                if (values[n] != 0) return values[n]; // تحقق من التخزين
                values[n] = Fibonacci(n - 1, values) + Fibonacci(n - 2, values); // حساب وتخزين القيمة
                return values[n];
    }
    static void Main()
    {
        int n = 6; // مثال عددي لحساب Fibonacci(6)
        int[] values = new int[n + 1]; // مصفوفة تخزين القيم المحسوبة مسبقًا
                int result = Fibonacci(n, values);
         Console.WriteLine($"Fibonacci({n}) = {result}");
    }
}
```

- **B. Bottom-Up (Tabulation)**

    1. Create a table (e.g., an array) to store the Fibonacci numbers.
    2. Initialize the base cases: F(0) = 0 and F(1) = 1.
    3. Iterate from 2 to 'n', computing each Fibonacci number using the formula F(i) = F(i-1) + F(i-2) and storing it in the table.
    4. The final result, F(n), is stored in the table at index 'n'.
    5. Explain how the bottom-up approach systematically builds the solution from the base cases.

```
using System;
class Program
{
    // Function to compute Fibonacci number
```

```
static int Fibi(int n)
{
    int past, prev, curr;
    past = prev = curr = 1; // curr holds Fib(i)
     for (int i = 2; i <= n; i++) // Compute next value
    {
        past = prev;
        prev = curr; // past holds Fib(i-2)
        curr = past + prev; // prev holds Fib(i-1)
    }
            return curr;
}
// Main method to test the Fibonacci function
static void Main()
{
    Console.Write("Enter a number to compute the Fibonacci: ");
    int n = int.Parse(Console.ReadLine());
    if (n == 0)
    {
        Console.WriteLine("Fibonacci of " + n + " is 0.");
    }
    else if (n == 1)
    {
        Console.WriteLine("Fibonacci of " + n + " is 1.");
    }
    else
    {
        int result = Fibi(n);
```

```
        Console.WriteLine("Fibonacci of " + n + " is " + result + ".");
    }}}
```

V. Time and Space Complexity Analysis

- Naive Recursive: Exponential time complexity (O(2^n)).
- Dynamic Programming (Both Top-Down and Bottom-Up): Linear time complexity (O(n)).
- Space Complexity: O(n) for both Top-Down (due to recursion stack) and Bottom-Up (due to the table).

VI. Applications of Dynamic Programming

- Briefly mention other classic dynamic programming problems, such as:
    - Knapsack Problem
    - Fibonacci Numbers
    - Matrix Chain Multiplication