Code:-

using System;

class Program

{

static double CalculateFutureValue(double presentValue, double rate, int years)

{

if (years == 0)

return presentValue;

return (1 + rate) \* CalculateFutureValue(presentValue, rate, years - 1);

}

static double CalculateFutureValueMemo(double presentValue, double rate, int years, double[] memo)

{

if (years == 0)

return presentValue;

if (memo[years] != 0)

return memo[years];

memo[years] = (1 + rate) \* CalculateFutureValueMemo(presentValue, rate, years - 1, memo);

return memo[years];

}

static void Main()

{

Console.WriteLine("=== Financial Forecasting Tool ===");

Console.Write("Enter present value (e.g., 1000): ");

double presentValue = Convert.ToDouble(Console.ReadLine());

Console.Write("Enter annual growth rate in % (e.g., 5): ");

double ratePercent = Convert.ToDouble(Console.ReadLine());

double rate = ratePercent / 100.0;

Console.Write("Enter number of years to forecast (e.g., 5): ");

int years = Convert.ToInt32(Console.ReadLine());

double futureValue = CalculateFutureValue(presentValue, rate, years);

Console.WriteLine($"\n[Recursive] Future value after {years} years: {futureValue:F2}");

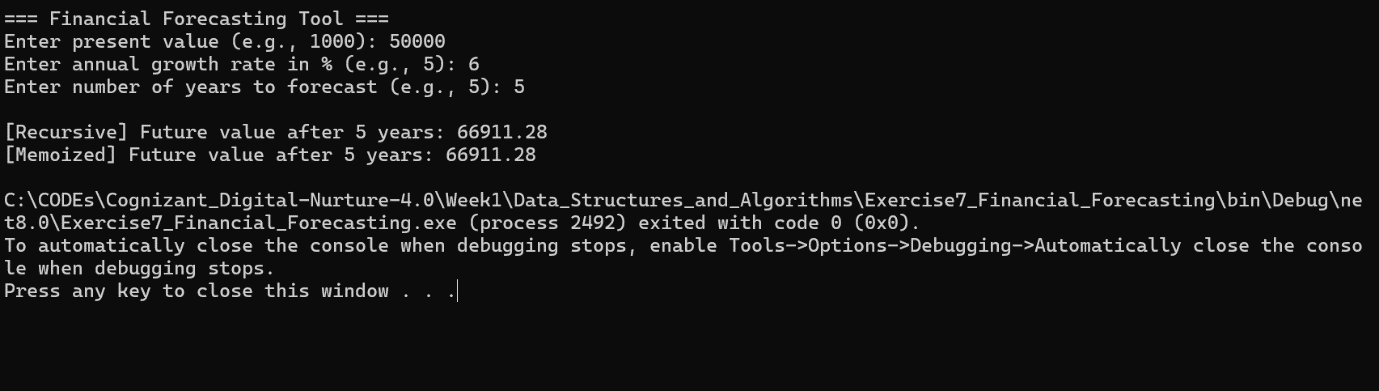
double[] memo = new double[years + 1];

double futureValueMemo = CalculateFutureValueMemo(presentValue, rate, years, memo);

Console.WriteLine($"[Memoized] Future value after {years} years: {futureValueMemo:F2}");

}

}



Q1. Understand Recursive Algorithms:

o Explain the concept of recursion and how it can simplify certain problems.

Answer:-

1. Understand Recursive Algorithms:

Recursion is a technique where a function calls itself to solve smaller subproblems of a larger problem.

It is especially useful for problems that exhibit overlapping subproblems and optimal substructure, such as

in tree traversal, factorials, and financial forecasts where each year's value depends on the previous year.

Each recursive function must have:

A base case: the simplest scenario where the recursion stops.

A recursive case: where the function calls itself with a smaller or simpler input.

🧠 How It Simplifies Problems

Recursion:

Breaks down complex problems into simpler subproblems.

Reduces the need for loops, making code cleaner and more elegant.

Is especially powerful in problems like:

Tree/graph traversal

Divide and conquer (e.g., merge sort, quicksort)

Mathematical computations (e.g., Fibonacci, factorial, compound interest)

💡 In the Context of Financial Forecasting:

Recursion can calculate future values year by year:

Each year's value depends on the previous year's value.

The recursive function keeps calling itself until it reaches year 0 (base case),

then builds the final result as the call stack unwinds.

/////////////////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////

Q4. Analysis:

o Discuss the time complexity of your recursive algorithm.

o Explain how to optimize the recursive solution to avoid excessive computation.

Answer:-

4.

a. Time Complexity of the Recursive Algorithm:

The recursive function for calculating future value is defined as:

futureValue(principal, rate, n) = futureValue(principal, rate, n - 1) \* (1 + rate)

Base Case: futureValue(principal, rate, 0) = principal

For n years, the function is called exactly n times, and each call performs a constant-time multiplication.

Therefore, the time complexity is:

Time Complexity: O(n)

Space Complexity (due to recursion stack): O(n)

b. Optimizing the Recursive Solution:

------------------------------------------------

---------------------------------------------

1. Convert to Iterative Approach:

To avoid stack overflow and improve performance, the recursive function can be rewritten as an iterative loop:

double futureValueIterative(double principal, double rate, int n) {

double result = principal;

for (int i = 0; i < n; i++) {

result \*= (1 + rate);

}

return result;

}

- Time Complexity: O(n)

- Space Complexity: O(1)

2. Use Mathematical Formula:

For a fixed annual growth rate, the future value can be computed directly using the compound interest formula:

Future Value = P \* (1 + r)^n

double futureValueFormula(double principal, double rate, int n) {

return principal \* Math.pow(1 + rate, n);

}

- Time Complexity: O(1)

- Space Complexity: O(1)

3. Use Memoization (Optional for Complex Scenarios):

If the same values are calculated repeatedly in complex recursive structures, memoization avoids redundant calculations:

Map<Integer, Double> memo = new HashMap<>();

double futureValueMemo(double principal, double rate, int n) {

if (n == 0) return principal;

if (memo.containsKey(n)) return memo.get(n);

double value = futureValueMemo(principal, rate, n - 1) \* (1 + rate);

memo.put(n, value);

return value;

}

- Time Complexity: O(n)

- Space Complexity: O(n)

Summary of Optimization Methods:

------------------------------------------------

| Method | Time Complexity | Space Complexity | Notes |

|--------------------|-----------------|------------------|----------------------------------------|

| Recursive | O(n) | O(n) | Simple but can cause stack overflow |

| Iterative | O(n) | O(1) | Safer and efficient for large inputs |

| Math Formula | O(1) | O(1) | Fastest and cleanest (for fixed rate) |

| Recursive + Memo | O(n) | O(n) | Useful for overlapping subproblems |