**Exercise 7: Financial Forecasting**

**What is a Recursive Algorithm?**

A **recursive algorithm** is an approach to solving a problem where the solution depends on solving smaller instances of the same problem. In programming, it is implemented by a **function that calls itself**, either directly or indirectly.

In essence, recursion breaks a complex problem into **simpler sub-problems** until it reaches a basic case that can be solved directly. These base cases stop the recursion and allow the solutions of the sub-problems to be combined to solve the original problem.

**Structure of a Recursive Function**

Every recursive function has two main parts:

1. **Base Case:**  
   The condition under which the recursion stops. It returns a direct answer without making another recursive call.
2. **Recursive Case:**  
   The part where the function calls itself with a smaller or simpler version of the original problem.

**Why Use Recursive Algorithms?**

Recursive algorithms are useful when:

* The problem has a **natural recursive structure** (e.g., trees, graphs, divide-and-conquer).
* The solution involves **repeating the same logic** on smaller inputs.
* They help **simplify the code**, making it cleaner and easier to understand.

**Common Applications of Recursion**

1. **Mathematical Computations**: Factorials, Fibonacci numbers, power, GCD

2. **Data Structures**: Tree traversal (preorder, inorder, postorder), graph search

3. **Divide and Conquer Algorithms**: Merge Sort, Quick Sort, Binary Search

4. **Backtracking**: Solving puzzles, generating permutations/combinations

5.**Dynamic Programming**: Recursion with memoization

**When to Use Recursion**

Use recursion when:

* The problem is naturally recursive (like trees or graphs).
* You need a concise solution and readability is more important than performance.
* The input size is small or limited.

## Time & Space Complexity

Recursive algorithms can be **inefficient** if not handled properly:

* **Time Complexity**: Depends on how many recursive calls are made.
  + E.g., Fibonacci recursion is exponential: O(2ⁿ)
* **Space Complexity**: Each recursive call uses stack space (O(n) in most simple cases).

**SourceCode:**

import java.text.DecimalFormat;

import java.util.HashMap;

import java.util.Map;

public class FinancialForecast {

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

public static double calculateFutureValue(double initialValue, double growthRate, int years) {

if (years == 0) return initialValue;

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

double result = calculateFutureValue(initialValue, growthRate, years - 1) \* (1 + growthRate);

memo.put(years, result);

return result;

}

public static void printForecast(double initialValue, double growthRate, int totalYears) {

DecimalFormat df = new DecimalFormat("#,##0.00");

System.out.println("========== Financial Forecast ==========");

System.out.println("Initial Investment: $" + df.format(initialValue));

System.out.println("Annual Growth Rate: " + (growthRate \* 100) + "%");

System.out.println("Forecast Period : " + totalYears + " years\n");

for (int year = 1; year <= totalYears; year++) {

double futureValue = calculateFutureValue(initialValue, growthRate, year);

System.out.println("Year " + year + ": $" + df.format(futureValue));

}

double finalValue = calculateFutureValue(initialValue, growthRate, totalYears);

System.out.println("\nFinal Forecasted Value After " + totalYears + " Years: $" + df.format(finalValue));

System.out.println("========================================");

}

public static void main(String[] args) {

double currentValue = 15000.0;

double annualGrowthRate = 0.07;

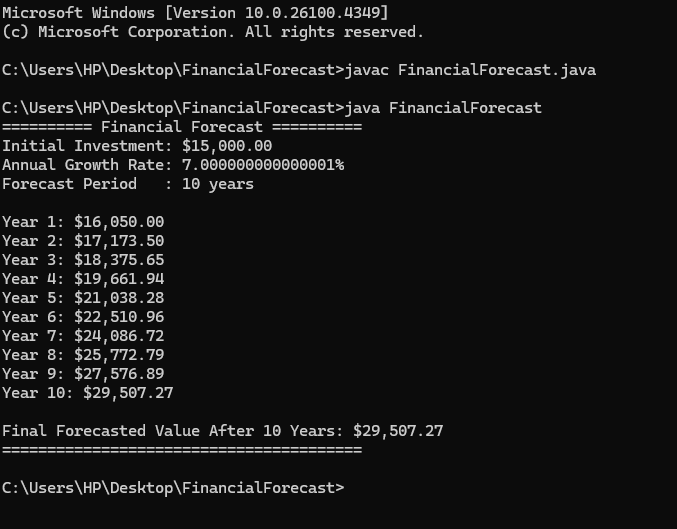
int forecastYears = 10;

printForecast(currentValue, annualGrowthRate, forecastYears);

}

}

**Output:**

****

**Analysis**

**Time Complexity**

The time complexity of the recursive algorithm is **O(n)**, where n is the number of years. This is because each recursive call handles one year and builds upon the result of the previous year. Without optimization, it may also involve redundant calculations.

**Optimization**

To avoid excessive computation, the solution is optimized using **memoization** — a technique where previously computed results are stored in a map. This prevents repeated calculations and significantly improves performance, especially for larger input sizes. Alternatively, an **iterative approach** can also be used to eliminate recursion overhead entirely

**Conclusion**

The recursive approach provides a clean and logical structure for forecasting financial growth year by year. When combined with memoization, it balances readability and performance, making it a practical and effective solution for small to medium forecasting tasks. For larger computations, iterative methods may offer better efficiency and stability.