**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

**Solutions:**

1. **Understanding:**

#### Concept of Recursion

* **Definition**: Recursion is a technique where a function calls itself in order to solve smaller instances of the same problem.
* **Base Case**: The condition under which the recursion stops. It prevents infinite recursion and ensures that the function eventually terminates.
* **Recursive Case**: The part of the function where it calls itself with a modified argument to move towards the base case.

**Advantages**:

* **Simplification**: Recursion can simplify the implementation of algorithms, especially for problems that naturally fit into a divide-and-conquer approach.
* **Readability**: Recursive solutions can be more intuitive and easier to understand for problems like tree traversal, factorial computation, and more.

**Disadvantages**:

* **Overhead**: Each recursive call adds a layer to the call stack, which can lead to stack overflow if the recursion depth is too high.
* **Performance**: Recursive algorithms can be less efficient due to redundant computations and excessive function calls.

1. **Setup and Implementation :**

public class FinancialForecasting { // Recursive method to calculate future value

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

// Base case: no more years left to apply the growth rate

if (years == 0) {

return presentValue;

} // Recursive case: apply the growth rate for one year and recurse for the remaining years

return calculateFutureValue(presentValue \* (1 + growthRate), growthRate, years - 1);

}

public static void main(String[] args) {

double presentValue = 1000; // Initial investment

double growthRate = 0.05; // Growth rate (5%)

int years = 10; // Number of years

double futureValue = calculateFutureValue(presentValue, growthRate, years);

System.out.println("Future Value after " + years + " years: $" + futureValue);

}

}

1. **Analysis:**

#### Time Complexity of Recursive Algorithm

* **Time Complexity**: O(n) - The function calculateFutureValue makes one recursive call for each year, resulting in a linear time complexity proportional to the number of years.

#### Optimization Techniques

**Memoization**: Store the results of previous computations to avoid redundant calculations. This technique is useful when the same subproblems are solved multiple times.

Code:-

import java.util.HashMap;

import java.util.Map;

public class FinancialForecastingWithMemoization {

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

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

// Check if result is in memo

if (memo.containsKey(years)) {

return memo.get(years);

}

// Base case

if (years == 0) {

return presentValue;

}

// Recursive case

double futureValue = calculateFutureValue(presentValue \* (1 + growthRate), growthRate, years - 1);

memo.put(years, futureValue); // Store result in memo

return futureValue;

}

public static void main(String[] args) {

double presentValue = 1000; // Initial investment

double growthRate = 0.05; // Growth rate (5%)

int years = 10; // Number of year

double futureValue = calculateFutureValue(presentValue, growthRate, years);

System.out.println("Future Value after " + years + " years: $" + futureValue);

}

}

**Iterative Approach**: For problems with simple recursion, an iterative approach using loops might be more efficient and avoids recursion depth issues.

Recursive algorithms provide a clear and elegant solution for problems that can be divided into similar sub problems, but they may suffer from inefficiencies for large inputs. Memoization and iterative solutions can help optimize performance and manage recursion depth.

Code:-

public class FinancialForecastingIterative {

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

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

presentValue \*= (1 + growthRate);

}

return presentValue;

}

public static void main(String[] args) {

double presentValue = 1000; // Initial investment

double growthRate = 0.05; // Growth rate (5%)

int years = 10; // Number of years

double futureValue = calculateFutureValue(presentValue, growthRate, years);

System.out.println("Future Value after " + years + " years: $" + futureValue);

}

}