**Exercise 7: Financial Forecasting**

**Code**

package com.example.forecasting;

public class FinancialForecast {

public static double predictFutureValue(double currentValue, double growthRate, int years) {

if (years == 0) {

return currentValue;

}

return *predictFutureValue*(currentValue \* (1 + growthRate), growthRate, years - 1);

}

public static double predictFutureValueMemo(double currentValue, double growthRate, int years, double[] memo) {

if (years == 0) {

return currentValue;

}

if (memo[years] != 0) {

return memo[years];

}

memo[years] = *predictFutureValueMemo*(currentValue \* (1 + growthRate), growthRate, years - 1, memo);

return memo[years];

}

public static void main(String[] args) {

double current = 10000;

double rate = 0.08;

int years = 5;

double future = *predictFutureValue*(current, rate, years);

System.*out*.printf("Predicted Future Value (Recursive): ₹%.2f%n", future);

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

double optimizedFuture = *predictFutureValueMemo*(current, rate, years, memo);

System.*out*.printf("Predicted Future Value (Memoized): ₹%.2f%n", optimizedFuture);

}

}

**Analysis**

Recursion is a programming technique where a function calls itself to solve smaller instances of a problem. In the context of financial forecasting, it simplifies the logic of compounding by modeling each year’s future value as dependent on the previous year's result. The basic recursive algorithm calculates the future value using the formula FV = PV \* (1 + r)^n, breaking it down into smaller yearly steps.

However, while recursive approaches are elegant and easy to understand, they can lead to **performance issues** due to redundant computations, especially for larger values of n. This results in a time complexity of **O(n)** and stack overhead from multiple recursive calls. To optimize, we use **memoization**, which stores results of already computed subproblems, preventing unnecessary recalculations. This turns the recursion into a more efficient, top-down dynamic programming solution. Memoization ensures that each unique subproblem is solved only once, greatly improving efficiency and reducing execution time. While the recursive approach works well for smaller inputs, optimized solutions using memoization or even iterative loops are preferred for scalability in real-world financial forecasting systems. This balance between simplicity and performance is key in algorithm design.