**Exercise 2:** **Financial Forecast**

**Code:**

***FinancialForecast.java***

package assignments.cts;

public class FinancialForecast {

//recursive solution

public static double futureValue(double principal, double rate, int years) {

if (years == 0) {

return principal;

}

return (1 + rate) \* *futureValue*(principal, rate, years - 1);

}

//iterative solution --> more optimal

public static double futureValueIterative(double principal, double rate, int years) {

double result = principal;

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

result \*= (1 + rate);

}

return result;

}

public static void main(String[] args) {

double principal = 10000;

double rate = 0.05;

int years = 10;

double futureVal = *futureValue*(principal, rate, years);

System.***out***.printf("Future value after %d years: %.2f\n --> recursive", years, futureVal);

System.***out***.println();

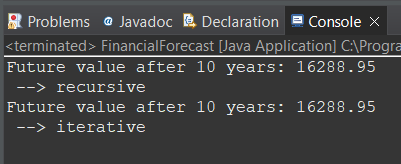
double futureValIter = *futureValueIterative*(principal, rate, years);

System.***out***.printf("Future value after %d years: %.2f\n --> iterative", years, futureValIter);

}

}

**Output:**

****

**Analysing of Recursive Solutions:**

**Recursion** is a programming technique where a function calls itself to solve a smaller instance of the same problem. It is especially useful for problems that have a natural recursive structure (e.g., factorial, Fibonacci series, tree traversal).

**Key concepts:**

* A **base case** stops the recursion.
* A **recursive case** calls the function again with a modified input.

**Why use recursion in financial forecasting?**

* Forecasting future values often involves applying a consistent transformation (e.g., growth) to previous values.
* Recursion allows us to define the future value in terms of previous ones.

**Recursive Formula:**

Assume:

* We have an initial value P (principal/investment).
* An annual growth rate r
* Number of years n.

The future value (FV) after n years is:

***FVn = P × (1 + r)n***

We can express this recursively as:

***FVn ​= (1 + r) × FVn – 1***

With a base case: ***FV0 = P***

**Time Complexity:**

* The function calls itself n times (from n down to 0).
* So, **Time Complexity = O(n)**

**Space Complexity:**

* Each recursive call adds a frame to the call stack.
* So, **Space Complexity = O(n)**

**Optimization:**

**Problem**: If n is large, this recursion could lead to a stack overflow.

**Optimized Approaches:**

* **Iteration**

**Code:**

public static double futureValueIterative(double principal, double rate, int years) {

double result = principal;

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

result \*= (1 + rate);

}

return result;

}