## Exercise 7: Financial Forecasting

### Step 1: Understand Recursive Algorithms

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

#### Recursion

* **Definition**: Recursion is a method of solving problems where a function calls itself as a subroutine. This allows the function to be repeated several times as it can call itself during its execution.
* **Usage**: Recursion is particularly useful for problems that can be broken down into smaller, similar subproblems. Common examples include factorial calculation, Fibonacci sequence, and tree traversals.
* **Base Case and Recursive Case**:
  + **Base Case**: The condition under which the recursion ends. Without a base case, the function would call itself indefinitely.
  + **Recursive Case**: The part of the function that includes the call to itself with modified arguments, bringing the function closer to the base case.

### Step 2: Setup

#### Define the Future Value Calculation Method

Let's assume we want to predict future values based on a constant growth rate.

// Java implementation

public class FinancialForecasting {

// Recursive method to calculate future value

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

// Base case

if (periods == 0) {

return presentValue;

}

// Recursive case

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

}

}

### Step 3: Implementation

#### Recursive Algorithm to Predict Future Values

// Java implementation

public class FinancialForecasting {

// Recursive method to calculate future value

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

// Base case

if (periods == 0) {

return presentValue;

}

// Recursive case

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

}

public static void main(String[] args) {

double presentValue = 1000.0;

double growthRate = 0.05; // 5% growth rate

int periods = 10;

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

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

}

}

### Step 4: Analysis

#### Time Complexity

* **Time Complexity**: The time complexity of this recursive algorithm is O(n), where n is the number of periods. This is because the function makes one recursive call for each period, leading to a linear number of calls.

#### Optimization to Avoid Excessive Computation

Recursion can lead to excessive computation, especially for large values of periods. To optimize the recursive solution, we can use **memoization** or **iteration**.

##### **Memoization**

Memoization involves storing the results of expensive function calls and reusing the stored results when the same inputs occur again.

// Java implementation with memoization

import java.util.HashMap;

public class FinancialForecasting {

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

// Recursive method with memoization to calculate future value

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

// Base case

if (periods == 0) {

return presentValue;

}

// Check memoization map

if (memo.containsKey(periods)) {

return memo.get(periods);

}

// Recursive case

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

memo.put(periods, futureValue);

return futureValue;

}

public static void main(String[] args) {

double presentValue = 1000.0;

double growthRate = 0.05; // 5% growth rate

int periods = 10;

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

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

}

}

##### **Iterative Approach**

An iterative approach can be more efficient and avoid the overhead of recursive calls and memoization.

// Java implementation with iteration

public class FinancialForecasting {

// Iterative method to calculate future value

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

double futureValue = presentValue;

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

futureValue \*= (1 + growthRate);

}

return futureValue;

}

public static void main(String[] args) {

double presentValue = 1000.0;

double growthRate = 0.05; // 5% growth rate

int periods = 10;

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

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

}

}

### Conclusion

* **Recursive Algorithm**: Provides a clear and straightforward approach, but can lead to excessive computation and stack overflow for large periods.
* **Memoization**: Optimizes the recursive approach by storing intermediate results, reducing redundant computations.
* **Iterative Approach**: Often more efficient for problems that can be solved using simple loops, avoiding the overhead of recursive calls and memoization.