**Exercise 7: Financial Forecasting**

**Scenario:**

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

1. Understand Recursive Algorithms:

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

**Recursion**: Recursion is a programming technique where a function calls itself to solve a problem by breaking it down into smaller, more manageable subproblems. The function continues to call itself with modified inputs until it reaches a base case, which is a simple scenario that can be solved directly without further recursion.

* Key components of recursion:
* Base case: The condition that stops the recursion
* Recursive case: The part where the function calls itself with a modified input
* **How recursion simplifies problems:**
* Divide and Conquer: Recursion allows us to break down complex problems into smaller, more manageable subproblems. This approach can make solving certain types of problems more intuitive and easier to understand.
* Elegant Solutions: For some problems, recursive solutions can be much more concise and elegant than their iterative counterparts. This often leads to cleaner, more readable code.
* Natural fit for certain problems: Some problems have a naturally recursive structure, such as tree traversals, fractals, or certain mathematical functions (e.g., factorial, Fibonacci sequence).
* Avoiding complex loop structures: Recursion can sometimes eliminate the need for complex nested loops or intricate iterative logic.
* Backtracking algorithms: Recursion is particularly useful in backtracking algorithms, where we need to explore multiple possibilities and undo choices**.**
* This recursive solution is more concise and arguably more intuitive than an iterative approach using loops.

However, it's important to note that while recursion can simplify certain problems, it's not always the most efficient solution. Recursive calls can consume more memory and potentially lead to stack overflow errors for very large inputs. In practice, the choice between recursive and iterative solutions often depends on the specific problem, performance requirements, and readability considerations.

2. Setup:

* + **Method Creation**: Create a method to calculate the future value using a recursive approach.

public static double calculateFutureValueRecursive(double initialValue, double[] growthRates, int futureYears, int currentYear) {

if (futureYears == 0) {

return initialValue;

}

double growthRate = growthRates[currentYear % growthRates.length];

double newValue = initialValue \* (1 + growthRate);

return calculateFutureValueRecursive(newValue, growthRates, futureYears - 1, currentYear + 1);

}

**calculateFutureValueRecursive()** Method:

* Takes the current value, the array of past growth rates, the number of years into the future, and the current year as parameters.
* Uses a recursive approach to calculate the future value:
  + Base case: If futureYears is 0, return the current value.
  + Recursive case: Calculate the new value using the current growth rate and call the method recursively with the updated parameters.

3. Implementation:

* + Implement a recursive algorithm to predict future values based on past growth rates.

public class FinancialForecasting {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.println("Enter the initial value:");

double initialValue = scanner.nextDouble();

System.out.println("Enter the number of past growth rates:");

int n = scanner.nextInt();

double[] growthRates = new double[n];

System.out.println("Enter the past growth rates (as percentages, e.g., 5 for 5%):");

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

growthRates[i] = scanner.nextDouble() / 100;

}

System.out.println("Enter the number of years into the future you want to predict:");

int futureYears = scanner.nextInt();

double futureValue = calculateFutureValueRecursive(initialValue, growthRates, futureYears, 0);

System.out.println("The predicted future value after " + futureYears + " years is: " + futureValue);

scanner.close();

}

**FinancialForecating** class - main() method:

* Prompts the user to enter the initial value.
* Prompts the user to enter the number of past growth rates and the rates themselves.
* Prompts the user to enter the number of years into the future for the prediction.
* Calls the calculateFutureValueRecursive method to calculate the future value and prints the result.

Recursive Algorithm:

* + Define a recursive function calculateFutureValueRecursive(initialValue, growthRates, futureYears, currentYear):
* **Base Case**: If futureYears is 0, return the initialValue.
* **Recursive Case**:
  + Calculate the current growth rate using the formula growthRate = growthRates[currentYear % growthRates.length].
  + Calculate the new value for the current year using newValue = initialValue \* (1 + growthRate).
  + Call the function recursively with newValue as the new initial value, futureYears - 1 as the number of future years left, and currentYear + 1 as the current year.
  + Return the result of the recursive function call
* Call the recursive function with the initial inputs to calculate the future value.

Here is the github repo link –

4. Analysis:

* + Discuss the time complexity of your recursive algorithm.

**Time Complexity:**

* Number of Recursive Calls: The function makes one recursive call for each year, so if futureYears is n, the function will make n recursive calls.
* Work Done per Call: In each call, the function performs a constant amount of work (calculating the growth rate, computing the new value).
  + Therefore, the total time complexity of the recursive function is O(n), where n is the number of years into the future (futureYears).
  + Explain how to optimize the recursive solution to avoid excessive computation.
* To optimize the recursive solution and avoid excessive computation, especially in the context of calculating future values, we can use techniques such as memoization or iterative approaches. Here’s how we can apply these optimizations:
* **Memoization**

Memoization is a technique where we store the results of expensive function calls and reuse those results when the same inputs occur again. For the future value calculation, memoization can help if the same futureYears and currentYear parameters are used repeatedly.

However, in this specific problem, memoization may not be straightforward due to the linear progression of years and the fact that each year depends on the previous year’s result. Instead, we will focus on an iterative approach, which is often more suitable for this kind of problem.

* **Iterative Approach**

Instead of using recursion, we can use an iterative approach with a loop to avoid the overhead of recursive calls and the potential for stack overflow with deep recursion.

public static double calculateFutureValueIterative(double initialValue, double[] growthRates, int futureYears) {

double currentValue = initialValue;

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

double growthRate = growthRates[i % growthRates.length];

currentValue \*= (1 + growthRate);

}

return currentValue;

}

* **Advantages of the Iterative Approach**
* **Avoids Stack Overflow**: Iterative approaches avoid the risk of stack overflow that can occur with deep recursion.
* **Better Performance**: Iterative solutions typically have lower overhead compared to recursive solutions due to the absence of recursive function calls.
* **Simpler Debugging**: Iterative code is often easier to debug and maintain compared to recursive code with complex stack traces.
* By using an iterative approach, we can optimize the future value calculation to be more efficient, with a time complexity of **O(n)** and constant space complexity (besides the input storage). This approach avoids the overhead of recursion and is generally more suitable for problems with a linear progression.

Output:

