**Exercise 7: Financial Forecasting**

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

Recursion is an algorithm where a function calls itself to solve smaller cases of the same problem. It is a mechanism for decomposing difficult problems into easier, more tractable sub-problems.

There are two components to any recursion problem –

1. Base Case

It is the condition that ends the recursion. It avoids infinite loops and gives a direct answer to the simplest instance of the problem.

1. Recursive Case

That portion of the function in which the problem is divided into smaller cases of the same problem.

Example:

public static int factorial(int n) {

if (n == 0 || n == 1) // Base case

return 1;

else // Recursive case

return n \* factorial(n - 1);

}

This function uses recursion to find the factorial of a given integer n. Here, n==0 or n==1 is the base case in which recursion stops and 1 is returned. n\*factorial(n-1) is the recursive case where the problem is broken down into smaller parts and the same function is called.

Advantages of recursion:

1. Divide and Conquer - Recursion decomposes a problem into smaller sub-problems, which are simpler to handle and solve. Every recursive call processes a less complicated version of the problem.
2. Recursive solutions tend to be more concise and readable than iterative solutions, particularly for problems with a natural recursive structure (e.g., tree traversal, calculating the factorial).
3. In problems whose solution is in stages or levels, recursion can make the problem-solving process easier by letting every function call solve a certain stage of the problem.

Disadvantages of recursion:

1. Stack Overflow - Excessive recursion will result in stack overflow errors when the depth of the recursion is excessively high. The reason behind this is that every function call creates a new layer on the call stack.
2. Performance Issues - Recursive programs might possess increased time complexity than iterative equivalents since calculation is duplicated. Methods such as memoization (storing results of recursive calls) assist in improving performance.
3. Debugging - Recursive solutions are sometimes more difficult to debug, particularly if there are logical bugs in the base case or recursive case.

**Discuss the time complexity of your recursive algorithm**

The recursive function calculateFutureValue calculates the compound future value by decrementing the number of years (years) until reaching zero, and then iterates upwards multiplying by (1 + growthRate) at every step.

When `years == 0`, the method returns the `presentValue` value, which is a constant time operation O(1).

In all other cases, the function calls itself n times until the base case is reached. Thus, the overall time complexity is O(n).

**Explain how to optimize the recursive solution to avoid excessive computation.**

There are two ways in which we can optimize this solution,

1. Memoization – When recursive calls are made for the same subproblems, storing the results can help directly fetch them instead of computing them again, and this is called memoization.

We can rewrite the program as the following –

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

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

if (years == 0) return presentValue;

if (cache.containsKey(years)) return cache.get(years);

double result = calculateFutureValueMemo(presentValue, growthRate, years - 1) \* (1 + growthRate);

cache.put(years, result);

return result;

}

1. Use an iterative approach – As there is no branching or overlapping in this program, we can also use an iterative approach instead of recursion.

We can write an iterative function as follows –

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

double futureValue = presentValue;

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

futureValue \*= (1 + growthRate);

}

return futureValue;

}

The time complexity for this iterative approach is also O(n). However, the space complexity is largely reduced from O(n) in the recursive approach to O(1) in this approach.

1. Use mathematical formula – Another method is to use the mathematical formula directly to calculate interest after a given number of years instead of calculating it iteratively or recursively.

For this, we can write the code as –

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

return presentValue \* Math.pow(1 + growthRate, years);

}

This takes constant time complexity, i.e., O(1)