# Exercise 7: Financial Forecasting

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

Recursion is a powerful concept in computer science where a function calls itself to solve a problem. Here the function keeps calling itself until it reaches a condition that stops the repetition, known as the *base case*.

Each recursive call breaks the problem into smaller, more manageable chunks. Once the base case is reached, the calls begin to resolve in reverse order, stacking back up to give the final result. It’s like divide and conquer then piling back to form the actual result .

Recursion simplifies certain problems by breaking them down into smaller, self-similar subproblems. This allows for elegant solutions to complex problems that might be difficult to solve iteratively. A recursive approach can make code more concise and readable, especially when dealing with problems that have a natural recursive structure.

Recursion Simplifies Problems by :

* + **Divide and Conquer:**

Recursion excels at breaking down large problems into smaller, manageable subproblems that are essentially the same problem but with simpler inputs.

* + **Code Clarity:**

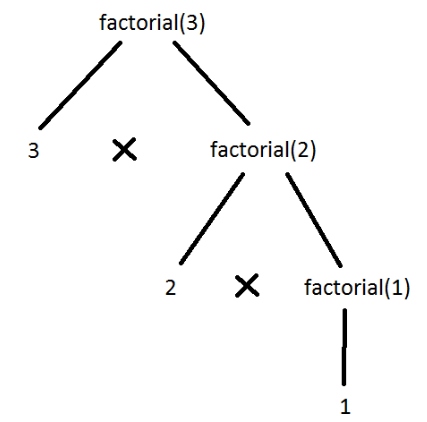
For problems with a natural recursive structure, a recursive solution can mirror the problem's structure, making the code easier to understand and follow.

* + **Conciseness:**

Recursive solutions can be more concise than iterative ones, particularly when dealing with problems that have nested or self-referential logic.

For example in the problem of factorial n! means =n\*n-1\*…\*1 therefore n-1 ! will be = n-1\*n-2\*…\*1  
 Note for n! has an n-1! Within it so we may write n! as n\*n-1! . This is what recursion takes into account because for every number its factorial is equal to that number multiplied by the previous number’s factorial Now what recursion does is whenever it encounters n! rather than calculating the same it divides the problem into two subparts – n and n-1! Just as I said before and again recursively apply factorial function on n-1 which is further divided into n-1 and n-2! And so on until it reaches 1! Which can be calculated in an unit time as 1.

So it forms a tree like structure and at last it again compiles up all the result to form the previous answer .

 p.c – wordpress.com

So as soon as we got 1! As 1 we can now calculate 2! Which is 2\*1! And after we got 2! By the same method we can calculate 3! Which is just the same as 3\*2! And so on and so forth.

That is how recursion simplifies the problem by diving the problem into smaller chunks and conquering or solving them and then compiling up to form the actual result .

4.Analysis

Discuss the time complexity of your recursive algorithm.

My algorithm of the given program calculates average of last year with the previous forecasted sales as follows:

public static double forecastSales(double[] sales, int years) {

if (years == 1) {

return sales[0];

}

return (sales[years - 1] + forecastSales(sales, years - 1)) / 2;

}

This is a simple **tail-recursive** algorithm where at each step I am:

* Calling the function once (forecastSales(sales, years - 1))
* Doing one addition and one division

**Time Complexity:**

* Each call reduces the problem size by 1.
* Making one recursive call per year (i.e., years times).
* Hence, **Time Complexity = O(n)**, where n = years

**Space Complexity:**

* Since it’s a recursive function with no memoization or array usage within recursion, and the call stack grows linearly with years:
* **Space Complexity = O(n)** (due to recursion stack)

Explain how to optimize the recursive solution to avoid excessive computation.  
  
Although my function is not inefficient but still it’s not totally optimal with respect to space complexity hence to make it optimal we must avoid the recursion as the recursion stack is taking up space and increasing the space complexity. Instead of recursion we may use iteration or dynamic programming where we can calculate the same sales as follows :

public static double forecastSalesOptimized(double[] sales, int years) {

double forecast = sales[0];

for (int i = 1; i < years; i++) {

forecast = (sales[i] + forecast) / 2;

}

return forecast;

}

Here the time complexity remains the same as O(n) as we are iterating through the sales array only once whereas the space complexity is reduced to O(1) as we are not using any extra space rather than a variable forecast which is again a constant space complexity.