**Exercise 7: Financial Forecasting**

**Scenario:**

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

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.

Recursion:

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. Using a recursive algorithm, certain problems can be solved quite easily.

A recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems. Many more recursive calls can be generated as and when required. It is essential to know that we should provide a certain case in order to terminate this recursion process. So we can say that every time the function calls itself with a simpler version of the original problem.

**How Recursion Works:**

* The function calls itself with a smaller input or a modified version of the original input.
* The function solves the smaller sub-problem and returns the result.
* The result is used to solve the original problem.
* The process repeats until the base case is reached.

1. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.

Time complexity of a recursive function depends on 2 factors.  
1. Total number of recursive calls  
2. Time complexity of additional operations for each recursive call.

The time complexity of the recursive algorithm is O(n). Like if a function calls itself two times then its time complexity is O(2 ^ N). if it calls three times then its time complexity is O(3 ^ N) and so on.

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

We can use a technique called **memoization** to save the computer time when making identical function calls. Memoization (a form of caching) remembers the result of a function call with particular inputs in a lookup table (the "memo") and returns that result when the function is called again with the same inputs.

A memoization of the factorial function could look like this:

* If \[n\] = 0, return 1
* Otherwise if \[n\] is in the memo, return the memo's value for \[n\]
* Otherwise,
  + Calculate \[(n - 1)! \times n\]
  + Store result in the memo
  + Return result

This algorithm checks for the input value in the memo *before* making a potentially expensive recursive call. The memo should be a data structure with efficient lookup times, such as a hash table with \[O(1)\] lookup time.