**WEEK1:Financial Forecasting**

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

Ans:

Recursion is a programming technique in which a function calls itself to solve a smaller instance of the same problem. Each recursive call works on a simpler **or** smaller input, gradually moving toward a base case, which stops the recursion. Once the base case is reached, the program returns values back up the chain of function calls, combining them to get the final result.

In financial forecasting, recursion can simplify problems that involve repeated patterns, time-based calculations, or hierarchical data, such as:

* Calculating compound interest over time,
* Forecasting profits for **f**uture years based on current growth trends,
* Or solving problems where the result depends on previous values, like predicting revenue using recurrence relations.

For example, if profit in a given year depends on the profits of the two previous years (like in a modified Fibonacci pattern), recursion can express this logic in just a few lines, without manually writing loops for each year. This makes recursive code cleaner, shorter, and easier to understand, especially for problems with natural repetitive structures.

However, while recursion can simplify logic, it should be used with care, as excessive recursion can lead to performance issues or stack overflow in cases with large input sizes. In such cases, memoization or converting to iteration is recommended.

2]Discuss the time complexity of your recursive algorithm.

Ans:

The time complexity of a recursive algorithm depends on how many times the function calls itself and how much work is done in each call. To analyze it, we usually express the number of recursive calls as a recurrence relation and solve it to find the overall complexity.

For example, consider a recursive function for financial forecasting, where the profit of the current year depends on the profits of the previous two years — similar to the Fibonacci series:

int forecastProfit(int year) {

if (year == 0) return 1000;

if (year == 1) return 1200;

return forecastProfit(year - 1) + forecastProfit(year - 2);

}

This recursive function makes two calls for each input year, leading to a time complexity of O(2ⁿ). It grows exponentially because each call branches into two more calls.

Such recursive algorithms are easy to write but become inefficient for large inputs. To improve performance, memoization or dynamic programming can be applied, which stores intermediate results and reduces the time complexity from O(2ⁿ) to O(n).

Therefore, while recursion simplifies the logic, its time complexity must be carefully analyzed to ensure efficiency, especially in data-intensive applications like financial forecasting.

3]Explain how to optimize the recursive solution to avoid excessive computation.

Ans:

Recursive algorithms are elegant and simple but can become inefficient if they repeat the same subproblems multiple times. This leads to excessive computation, especially in problems like financial forecasting where each calculation may depend on several earlier results, such as in compound interest or multi-year profit predictions.

To optimize a recursive solution, we can use the following techniques:

1. Memoization (Top-Down Dynamic Programming):Memoization involves storing the results of recursive calls in a data structure like an array or a map so that they don’t need to be recomputed.

* Instead of recalculating the profit for year 3 multiple times, we store it the first time it is computed and reuse the value.
* This reduces time complexity from exponential (O(2ⁿ)) to linear (O(n)).

Example: int[] memo = new int[100];

int forecastProfit(int year) {

if (year == 0) return 1000;

if (year == 1) return 1200;

if (memo[year] != 0) return memo[year];

memo[year] = forecastProfit(year - 1) + forecastProfit(year - 2);

return memo[year];

}

2. Tabulation (Bottom-Up Dynamic Programming):Instead of solving the problem recursively, we build the solution iteratively from the base case up to the final year.

* This avoids the function call stack entirely and saves more memory.
* It’s often faster and more space-efficient than memoization.

int forecastProfit(int year) {

int[] dp = new int[year + 1];

dp[0] = 1000;

dp[1] = 1200;

for (int i = 2; i <= year; i++) {

dp[i] = dp[i - 1] + dp[i - 2];

}

return dp[year];

}

3. Using Tail Recursion (where possible):In some languages (like functional languages), using tail recursion allows the compiler to optimize recursive calls and avoid stack overflow.