**Exercise 7: Financial Forecasting**

**I. Understanding Recursive Algorithms**

Recursion is a way of solving problems where a function calls itself to work through smaller pieces of the same task. Think of it like solving a puzzle—breaking it into smaller parts until it's easy to handle.

Every recursive function has two important parts:

* Base Case: This is the stopping point. It tells the function when to stop calling itself.
* Recursive Case: This is where the function keeps calling itself with simpler data until it reaches the base case.

In financial forecasting, we often need to apply the same growth rate over and over—like figuring out what your investment will be worth each year. Recursion fits nicely here because each year's value depends on the one before it. You start with the initial amount and keep building on it, year after year.

**IV. Evaluation**

Time Complexity

This type of recursive forecasting takes O(n) time, where *n* is the number of years you're predicting. That's because it makes one function call per year until it hits the base case.

Space Complexity

Each function call takes up memory, so the space it uses also grows with the number of years—O(n) space.

How to Make It Better

While recursion works well and looks clean, it’s not always the most efficient for bigger inputs. Why?

* Too many recursive calls can cause a stack overflow.
* It repeats similar calculations, wasting time.

To make it better:

* Memoization: Save results from earlier calculations so you don’t have to redo them.
* Use a Loop Instead: An iterative approach (with a simple loop) can do the same job with less memory and better performance.

**Exercise 2: E-commerce Platform Search Function**

**I. Understanding Asymptotic Notation**

When building search features in apps—like an online shopping site—it’s important to understand how fast your code runs as the number of products grows. That’s where asymptotic notation comes in. It helps us describe how an algorithm behaves when the input size increases.

What is Big O?

Big O tells us the worst-case time your code might take. It’s like planning for traffic during rush hour—what’s the slowest it could be?

Different Cases to Think About:

* Best Case: The fastest scenario (e.g., finding the item right at the top of the list).
* Average Case: What usually happens in normal conditions.
* Worst Case: The slowest case (e.g., the item is missing or last in the list).

Knowing these helps you choose better algorithms and make sure your app stays fast—even when it has to search through thousands of products.

**IV. Evaluation**

Time Complexity

* Linear Search: Takes O(n) time. It checks every item one by one until it finds the right one—or finishes the whole list.
* Binary Search: Much faster at O(log n) time. It splits the list in half each time, but it only works if the list is already sorted.

Space Complexity

Both methods are memory-efficient and use only O(1) space, since they don’t create new data structures.

Which is Better?

Binary search is the better choice if your product list is sorted. It’s fast and scales well for big catalogs.

But linear search still has its place:

* It works fine for small lists.
* It’s easier when sorting the data isn’t practical.
* It’s simple and quick to implement when you don’t search often.

How to Improve Search Performance

* Keep your product list sorted and use binary search whenever possible.
* If you’re doing a lot of searches, consider using hash maps or search trees for instant lookups.
* Choose the search method that fits the size and needs of your platform.