Exercise 2: E-commerce Platform Search Function

1. Understanding Asymptotic Notation

Big O Notation:
Big O notation is a way to describe how efficient an algorithm is in terms of time or space as the input size increases. It gives us an idea of how well an algorithm scales. Instead of focusing on actual execution time, it helps us analyze the *growth* of an algorithm's performance. For example, an algorithm with O(n) complexity means the time taken increases linearly with the number of elements.

Best, Average, and Worst Cases:
When we talk about search operations, we often look at three scenarios:

Best Case: The item is found immediately (like the first element). This is the fastest scenario.

Average Case: The item is somewhere in the middle, so we do a few comparisons before finding it.

Worst Case: The item is not in the list or is at the end, so we have to check all elements.

These cases help us understand how the algorithm performs in real situations—not just in ideal conditions.

4. Analysis

When comparing both searches:

Linear Search takes O(n) time in the worst case. This means if there are 1,000 products, it might have to check all 1,000.

Binary Search is much faster with O(log n) time, but the list has to be sorted first.

Exercise 7: Financial Forecasting

1. Understand Recursive Algorithms

What is Recursion?
Recursion is a programming technique where a function calls itself to solve a smaller instance of the same problem. It's like breaking a big problem into smaller and more manageable chunks. The idea is to keep reducing the problem until you reach a simple case (called the **base case**) that can be solved directly.

For example, calculating the factorial of a number (n!) or finding the nth number in the Fibonacci sequence can be naturally expressed with recursion. It often makes the code cleaner and more readable for problems that have a repetitive structure.

4. Analysis

Time Complexity:
The time complexity of this recursive approach is **O(n)**, where *n* is the number of years.
That's because the function calls itself once for each year, reducing the number of years by 1 each time until it reaches 0.

Optimization:
While the time complexity is linear and may seem fine, recursion can still lead to **excessive memory usage** due to the function call stack, especially if the number of years is large.

To optimize:

> We can **convert it to an iterative solution** (which is more memory-efficient).
>
> Or, in some recursive problems where repeated calculations happen (like in Fibonacci), we can use **memoization** (caching results) to avoid redundant work.

In our case, since each step depends only on the previous year, memoization isn't necessary—but switching to iteration for large inputs can prevent stack overflow.