**ALGORITHMS\_DATA STRUCTURES**

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

1. **Explain Big O notation and how it helps in analyzing algorithms.**

**Ans)** Big O notation is used to describe the performance or complexity of an algorithm in terms of time or space as the input size increases. It provides an upper bound on the growth rate of an algorithm and is essential for analyzing the efficiency and scalability of code.

Big O ignores constants and lower-order terms to focus on the most significant factors that affect performance as input size grows.

**Q)Describe the best, average, and worst-case scenarios for search operations.**

**Ans)** We will consider 3 types of performance cases:-

**Linear Search**

- Best Case: O(1) — The element is found at the beginnin

- Average Case: O(n) — The element is somewhere in the middle.

- Worst Case: O(n) — The element is at the end or not present.

Linear search checks each item one by one. It is simple but inefficient for large datasets.

**Binary Search**

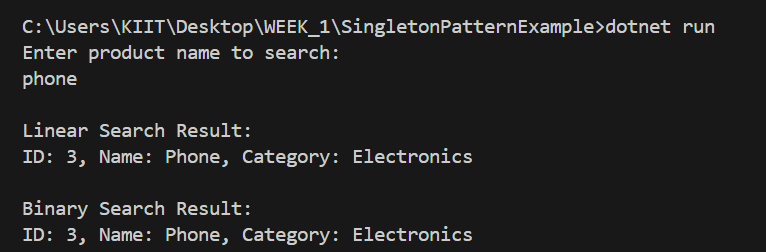
- Best Case: O(1) — The middle element is the target.

- Average Case: O(log n) — The element is found after a few divisions.

- Worst Case: O(log n) — The element is not found after log₂(n) steps.

Binary search is much faster than linear search, but it requires the input array to be sorted in advance.

**CODE OUTPUT ScreenShot:-**



**STEP 4 Analysis:**

**Q)Compare the time complexity of linear and binary search algorithms.**

**Ans)** Linear Search and Binary Search are two fundamental algorithms used to locate an element within a collection. Their time complexities differ significantly in terms of efficiency and applicability.

**Linear Search** operates by sequentially checking each element of the array until the desired item is found or the end of the list is reached.

Best Case Time Complexity: O(1) — when the element is at the first position.

Average Case Time Complexity: O(n) — when the element is somewhere in the middle.

Worst Case Time Complexity: O(n) — when the element is at the end or not present.

**Binary Search** works by repeatedly dividing a sorted array into halves and comparing the middle element with the target.

Best Case Time Complexity: O(1) — when the middle element is the target.

Average Case Time Complexity: O(log n) — efficient even with larger datasets.

Worst Case Time Complexity: O(log n) — element is not present after multiple splits.

The key difference lies in the fact that binary search requires the data to be sorted in advance, while linear search works on both sorted and unsorted data but is significantly less efficient for large datasets.

**Q)Discuss which algorithm is more suitable for your platform and why.**

**Ans) Binary search is more suitable for an e-commerce platform** because it offers significantly better performance compared to linear search when handling large datasets.

E-commerce systems typically deal with thousands of products and require real-time response during searches. Binary search has a time complexity of O(log n), making it much faster than linear search (O(n)), especially as the number of products grows.

The only requirement for binary search is that the data must be sorted. In modern applications, maintaining sorted data (e.g., by product name or ID) is manageable and can be automated.

For small-scale systems or unsorted data, linear search may be used temporarily. But for any serious production-level e-commerce application, binary search is the right approach.

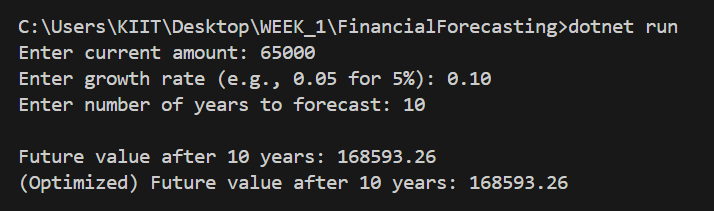
**Exercise 7: Financial Forecasting**

**Q)Explain the concept of recursion and how it can simplify certain problems.**

**Ans)** Recursion is a technique where a method calls itself to solve a problem. It is often used to break down a complex problem into simpler sub-problems. Recursion can simplify problems that have a natural hierarchical structure or repetitive steps.

In the context of financial forecasting, recursion can be used to calculate the future value of an investment based on a fixed growth rate over time.

**CODE OUTPUT ScreenShot:-**



**Q)Discuss the time complexity of your recursive algorithm.**

**Ans)** The recursive algorithm used to calculate future value follows a simple linear recursive structure, where the function calls itself once for each year until it reaches the base case.

Method -

static double PredictFutureValue(double amount, double growthRate, int years)

    {

        if (years == 0)

            return amount;

        return PredictFutureValue(amount \* (1 + growthRate), growthRate, years - 1);

    }

Time Complexity:

Time Complexity: O(n), where n is the number of years.

Explanation: Each recursive call represents one year of growth. For n years, the function will make n recursive calls.

Space Complexity: Also O(n) due to the recursion stack, which can grow linearly with the number of years.

This makes the recursive approach less efficient for large values of n, as it may lead to stack overflow errors or performance degradation.

**Q)Explain how to optimize the recursive solution to avoid excessive computation**

**Ans) To avoid excessive computation and potential stack overflow caused by deep recursion, the recursive solution can be replaced with a direct mathematical formula or an iterative approach.**

**Optimized Approach Using Math:**

double PredictFutureValueOptimized(double amount, double growthRate, int years)

{

    return amount \* Math.Pow(1 + growthRate, years);

}

**Advantages:**

**Time Complexity: O(1) – Constant time, regardless of the number of years.**

**Space Complexity: O(1) – No additional memory usage from recursion.**

**Performance: Significantly faster and more scalable for large time periods.**

**Alternatively, if you must avoid recursion but want to simulate it, an iterative loop can be used instead of recursive calls.**

**Iterative version:**

double PredictFutureValueIterative(double amount, double growthRate, int years)

{

    for (int i = 0; i < years; i++)

    {

        amount \*= (1 + growthRate);

    }

    return amount;

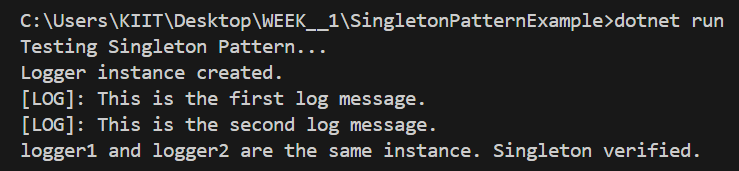
}

**This approach avoids recursion entirely while maintaining clarity and performance.**

****DESIGN PATTERNS AND PRINCIPLES****

****Exercise 1: Implementing the Singleton Pattern****

****Code Output ScreenShot:-****



**Exercise 2: Implementing the Factory Method Pattern**

****Code Output ScreenShot:-****

