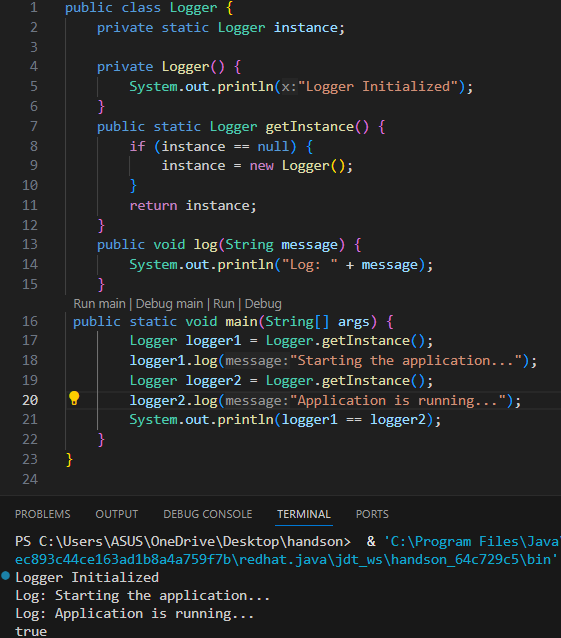
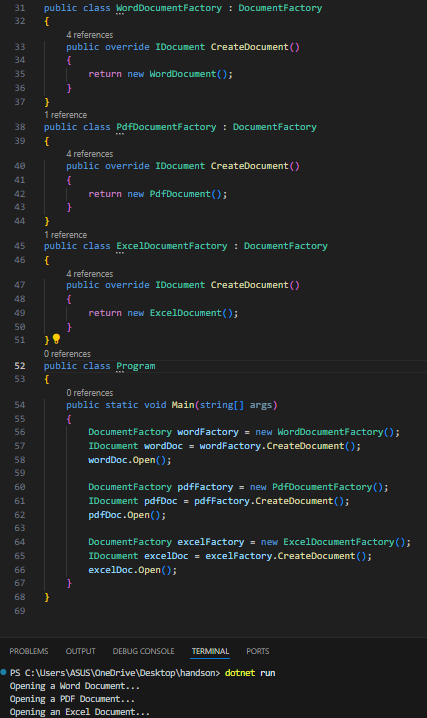
**HANDSON**

**Module1-Design Patterns and Principles**

**Exercise 1: Implementing the Singleton Pattern**

****

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

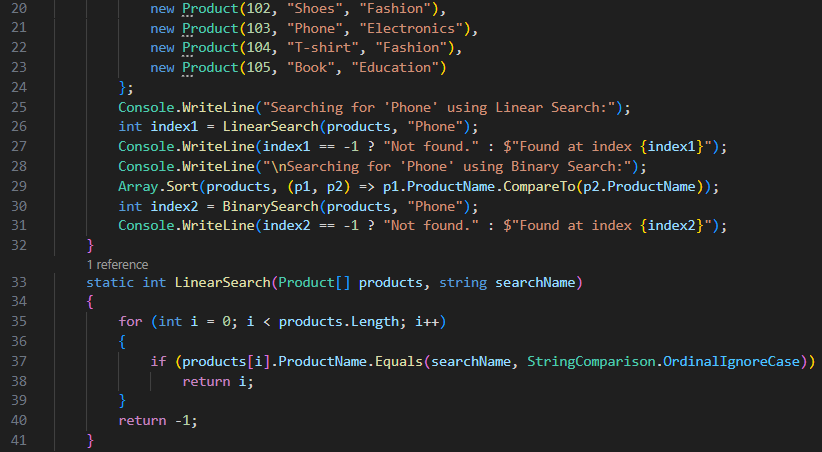
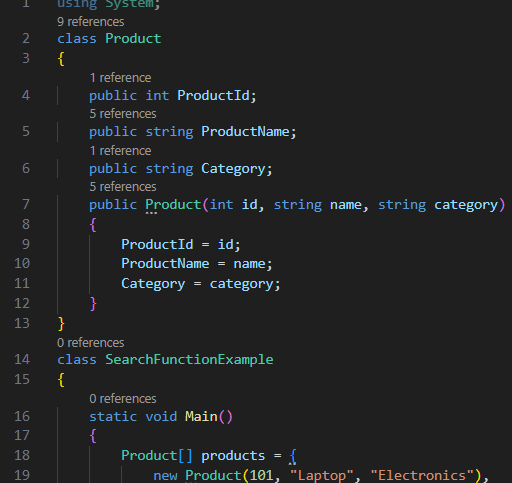
**Module2-** **Data structures and Algorithms**

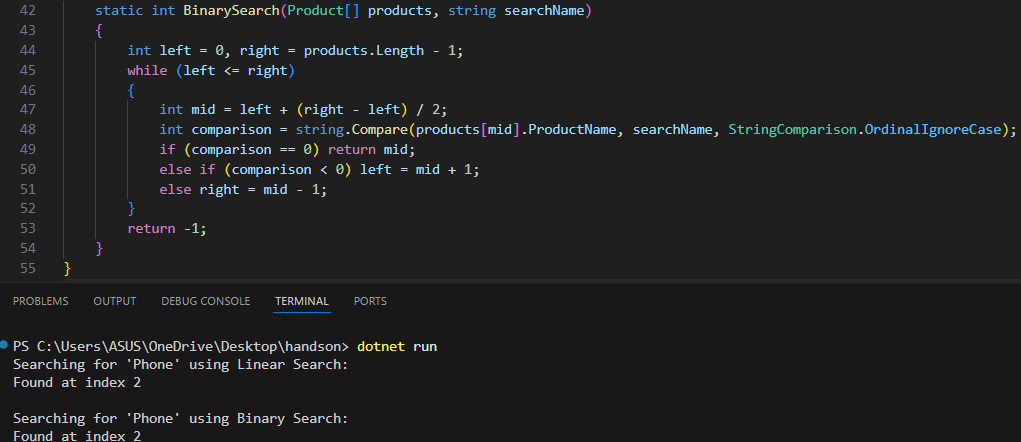
**Exercise 3: E-commerce Platform Search Function**

Big O Notation:Big O describes how fast or slow an algorithm is as input size (n) grows.

Linear Search → O(n) → Time grows linearly with data size.Binary Search → O(log n) → Time grows logarithmically → much faster for large data.

| ***Case*** | ***Linear Search*** | ***Binary Search*** |
| --- | --- | --- |
| ***Best*** | *O(1)* | *O(1)* |
| ***Average*** | *O(n/2) ≈ O(n)* | *O(log n)* |
| ***Worst*** | *O(n)* | *O(log n)* |

****

****

| Algorithm | Time Complexity | Suitable For |
| --- | --- | --- |
| Linear Search | O(n) | Small datasets / unsorted data |
| Binary Search | O(log n) | Large datasets / sorted data |

* Binary Search is better for large, sorted datasets
* Linear Search works for small or unsorted data

**Exercise 4: Financial Forecasting**

Recursion is when a function calls itself to solve a smaller version of the same problem.

It continues doing that until a stop condition (called the base case) is met.

Recursion is useful when:

1) A problem can be broken into smaller, similar problems.

2) Reduces the number of lines in code by reusing

Ex: the below code call the function itself again and again until base condition i.e n==1 occure

int Factorial(int n)

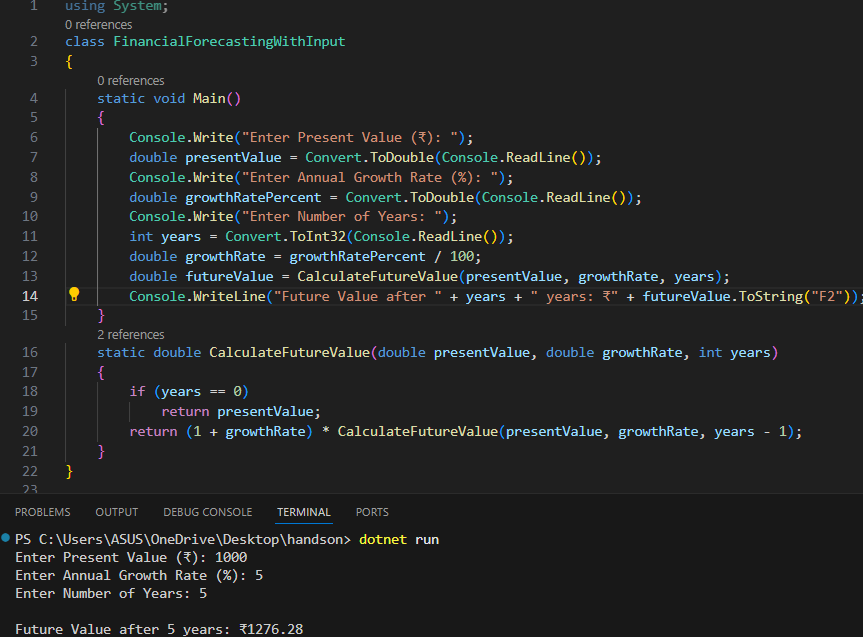
{

if (n == 1)

return 1

return n \* Factorial(n - 1);

}



The time complexity of this recursive algorithm is O(n), where n represents the number of years we want to predict. This is because for each year, the function makes one recursive call, leading to a total of n calls by the time it finishes. Each of these calls only performs a couple of simple operations like multiplication and subtraction, which are done in constant time. So overall, the time taken grows linearly with the number of years.

When it comes to space complexity, it’s also O(n). This happens because each recursive call is stored on the call stack until the entire chain of recursion is complete. The problem with this is that if we try to predict for a very large number of years, the stack can fill up, leading to a stack overflow error. This is why, for large inputs, we usually prefer using an iterative or optimized approach to avoid these risks.

Optimization recommended

**Iterative Approach**

static double CalculateFutureValueIterative(double presentValue, double growthRate, int years) { double result = presentValue;

for (int i = 0; i < years; i++) result \*= (1 + growthRate);

return result;

}

Time Complexity =O(n) since the loop runs no.of years input i.e n  
Space Complexity =O(1) since memory does not grow with input vary

**Math.Pow()** **Approach (**Best choice since No recursion, no loops, fastest result**)**

static double CalculateFutureValueOptimal(double presentValue, double growthRate, int years) {

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

}

Time Complexity = O(1) since no loop or recursion,direct return  
Space Complexity = O(1) no memory grow based on input vary