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

Big O notation is a mathematical notation used to describe the time or space complexity of an algorithm in terms of input size (n). It helps developers understand the scalability and efficiency of algorithms by focusing on their growth rate, not exact execution time.

Search Scenarios:

* Best Case: The item is found in the first few comparisons.
* Average Case: The item is found somewhere in the middle.
* Worst Case: The item is not present or is at the last position (linear) or requires full log(n) comparisons (binary).

**Code**

using System;

namespace ECommerceSearch

{

public class Product

{

public int productId;

public string productName;

public string category;

public Product(int id, string name, string category)

{

productId = id;

productName = name;

this.category = category;

}

public override string ToString()

{

return $"ID: {productId}, Name: {productName}, Category: {category}";

}

}

class Searchtype

{

static Product LinearSearch(Product[] products, string name)

{

foreach (var product in products)

{

if (product.productName.Equals(name, StringComparison.OrdinalIgnoreCase))

return product;

}

return null;

}

static Product BinarySearch(Product[] products, string name)

{

int left = 0, right = products.Length - 1;

while (left <= right)

{

int mid = (left + right) / 2;

int comparison = string.Compare(products[mid].productName, name, StringComparison.OrdinalIgnoreCase);

if (comparison == 0)

return products[mid];

else if (comparison < 0)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

static void Main(string[] args)

{

Product[] products = {

new Product(101, "Bag", "Accessories"),

new Product(102, "Camera", "Electronics"),

new Product(103, "Laptop", "Electronics"),

new Product(104, "Mouse", "Electronics"),

new Product(105, "Shoes", "Footwear")

};

Array.Sort(products, (p1, p2) => string.Compare(p1.productName, p2.productName, StringComparison.OrdinalIgnoreCase));

Console.WriteLine("Linear Search for 'Laptop':");

var linearResult = LinearSearch(products, "Laptop");

Console.WriteLine(linearResult != null ? linearResult.ToString() : "Not Found");

Console.WriteLine("\nBinary Search for 'Laptop':");

var binaryResult = BinarySearch(products, "Laptop");

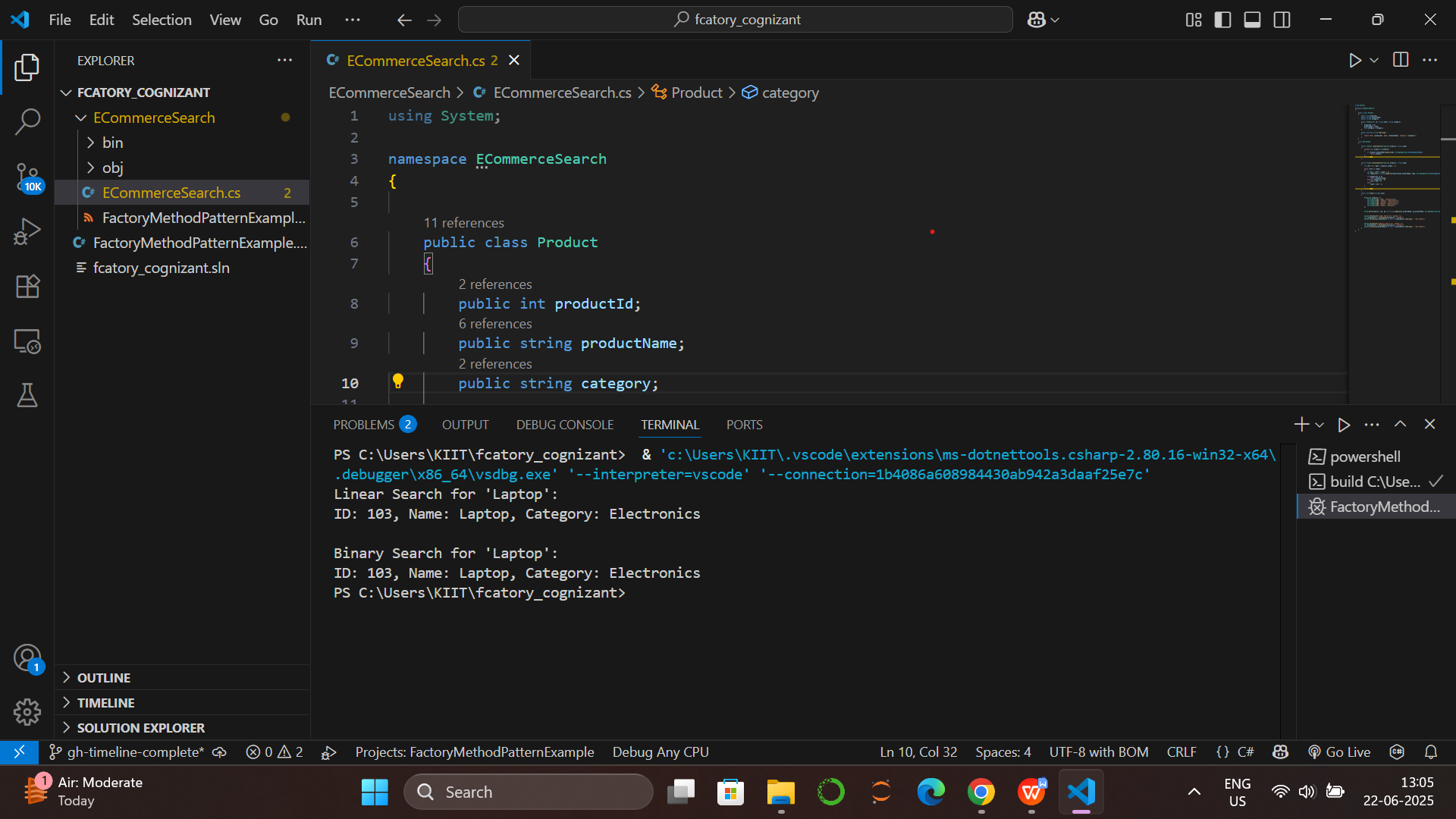
Console.WriteLine(binaryResult != null ? binaryResult.ToString() : "Not Found");

}

}

}

**Output**



Linear Search:

* Best Case: The item is found at the first position → O(1)
* Average Case: The item is found around the middle of the list → O(n)
* Worst Case: The item is not found or is at the last position → O(n)

Binary Search :

* Best Case: The item is found at the middle of the array → O(1)
* Average Case: The item is found after a few divisions → O(log n)
* Worst Case: The item is not found and all divisions are performed → O(log n)

**Binary search** is more suitable for an e-commerce platform where fast search is needed and data can be kept sorted.

**Exercise 7: Financial Forecasting**

Recursion is a method where a function calls itself to solve smaller instances of a problem. It is useful for problems like computing factorial, Fibonacci, and in this case, future value prediction based on compound growth.

**Code:**

using System;

namespace FinancialForecasting

{

class Program

{

static void Main(string[] args)

{

double initialAmount = 1000; // Starting value (e.g., initial investment)

double growthRate = 0.05; // 5% growth rate

int years = 10; // Forecasting for 10 years

Console.WriteLine("Recursive Future Value Prediction");

double futureValueRecursive = CalculateFutureValueRecursive(initialAmount, growthRate, years);

Console.WriteLine($"Future Value (Recursive): {futureValueRecursive:F2}");

Console.WriteLine("\nOptimized (Memoized) Future Value Prediction");

var memo = new double?[years + 1];

double futureValueMemo = CalculateFutureValueMemo(initialAmount, growthRate, years, memo);

Console.WriteLine($"Future Value (Memoized): {futureValueMemo:F2}");

}

static double CalculateFutureValueRecursive(double amount, double rate, int years)

{

if (years == 0)

return amount;

return CalculateFutureValueRecursive(amount, rate, years - 1) \* (1 + rate);

}

static double CalculateFutureValueMemo(double amount, double rate, int years, double?[] memo)

{

if (years == 0)

return amount;

if (memo[years].HasValue)

return memo[years].Value;

memo[years] = CalculateFutureValueMemo(amount, rate, years - 1, memo) \* (1 + rate);

return memo[years].Value;

}

}

}

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

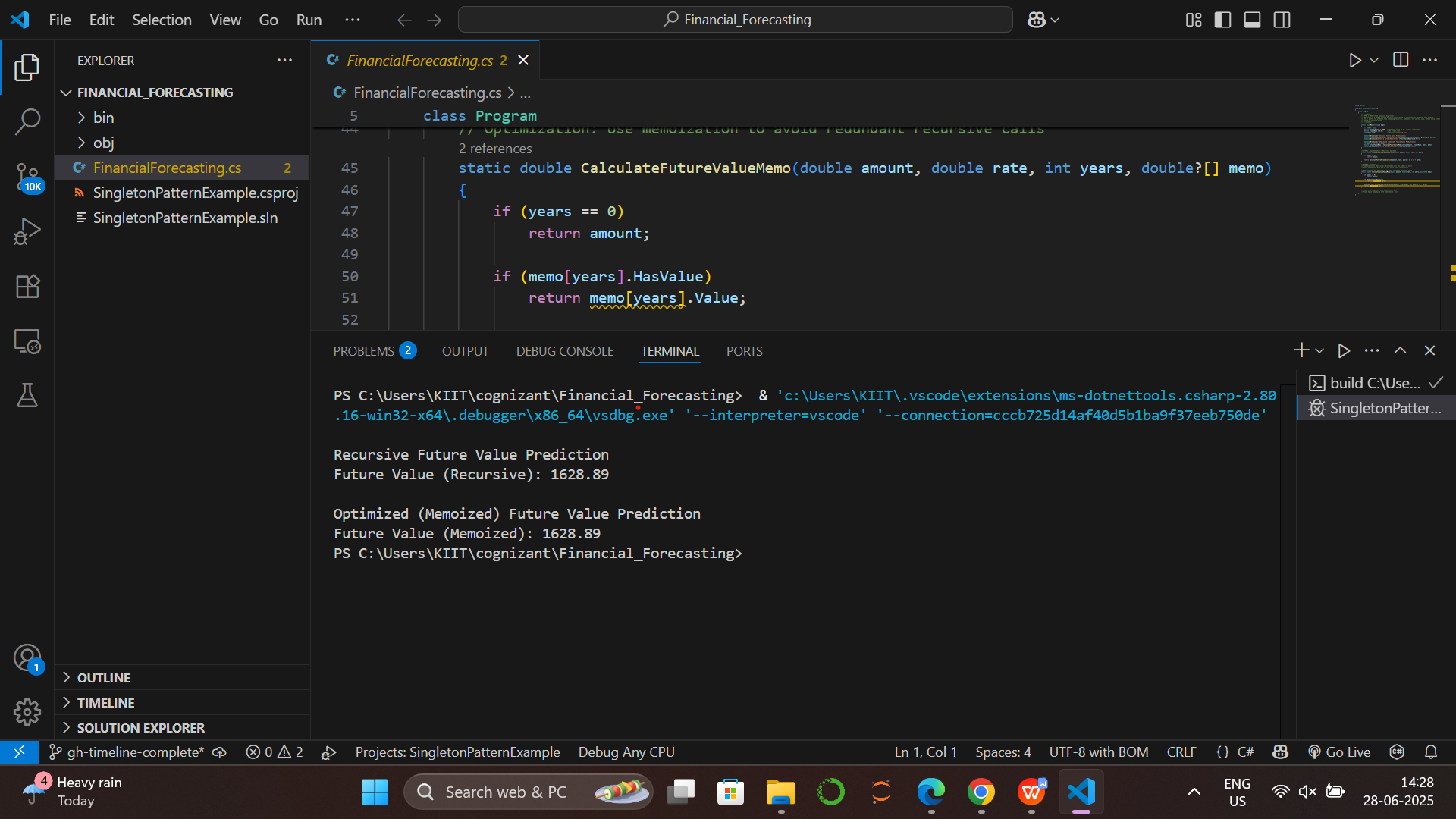
Space Complexity: O(n) due to call stack usage in recursion.

Optimization: Use memoization to avoid redundant recursive calls

Final Time Complexity with Memoization: O(n)

Final Space Complexity with Memoization: O(n)

**Output:**



**Exercise 1: Inventory Management System**

Data structures and algorithms are crucial in an inventory management system because they determine how efficiently data is **stored, accessed, modified, and deleted**.

In an inventory management system, the choice of data structure directly affects performance, especially for operations like search, update, and delete:

1. **Dictionary**: Best for fast lookup, add, update, and delete operations using product ID.
2. **List**: Suitable for maintaining ordered products but slower for search and update.
3. **SortedDictionary**: Maintains sorted order of products and allows efficient lookups.
4. **Queue/Stack**: Useful for handling inventory operations in FIFO or LIFO order.
5. **Trie**: Ideal for fast prefix-based searches like autocomplete on product names.

**Code:**  
using System;

using System.Collections.Generic;

namespace InventorySystem

{

public class Product

{

public int productId { get; set; }

public string productName { get; set; }

public int quantity { get; set; }

public double price { get; set; }

public Product(int productId, string productName, int quantity, double price)

{

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

}

public class InventoryManager

{

private Dictionary<int, Product> inventory = new Dictionary<int, Product>();

public void AddProduct(Product product)

{

if (!inventory.ContainsKey(product.productId))

{

inventory[product.productId] = product;

Console.WriteLine("Product added.");

}

else

{

Console.WriteLine("Product already exists.");

}

}

public void UpdateProduct(int productId, int quantity, double price)

{

if (inventory.ContainsKey(productId))

{

inventory[productId].quantity = quantity;

inventory[productId].price = price;

Console.WriteLine("ProductID " +productId + " - Product updated.");

}

else

{

Console.WriteLine("Product not found.");

}

}

public void DeleteProduct(int productId)

{

if (inventory.Remove(productId))

{

Console.WriteLine("ProductID " +productId + " - Product deleted.");

}

else

{

Console.WriteLine("Product not found.");

}

}

public void DisplayInventory()

{

Console.WriteLine("\n Current Inventory:");

foreach (var product in inventory.Values)

{

Console.WriteLine("ID: " + product.productId + ", Name: " + product.productName + ", Quantity: "

+ product.quantity + ", Price: " + product.price);

}

}

}

class Program

{

static void Main(string[] args)

{

InventoryManager manager = new InventoryManager();

manager.AddProduct(new Product(301, "Laptop", 25, 75000));

manager.AddProduct(new Product(302, "Smartphone", 30, 20000));

manager.AddProduct(new Product(303, "Tablet", 15, 17000));

Console.WriteLine("All Products are added successfully.");

manager.DisplayInventory();

Console.WriteLine("\n");

manager.UpdateProduct(302, 15, 25000);

manager.DeleteProduct(303);

Console.WriteLine("After updating and deleting product from inventory");

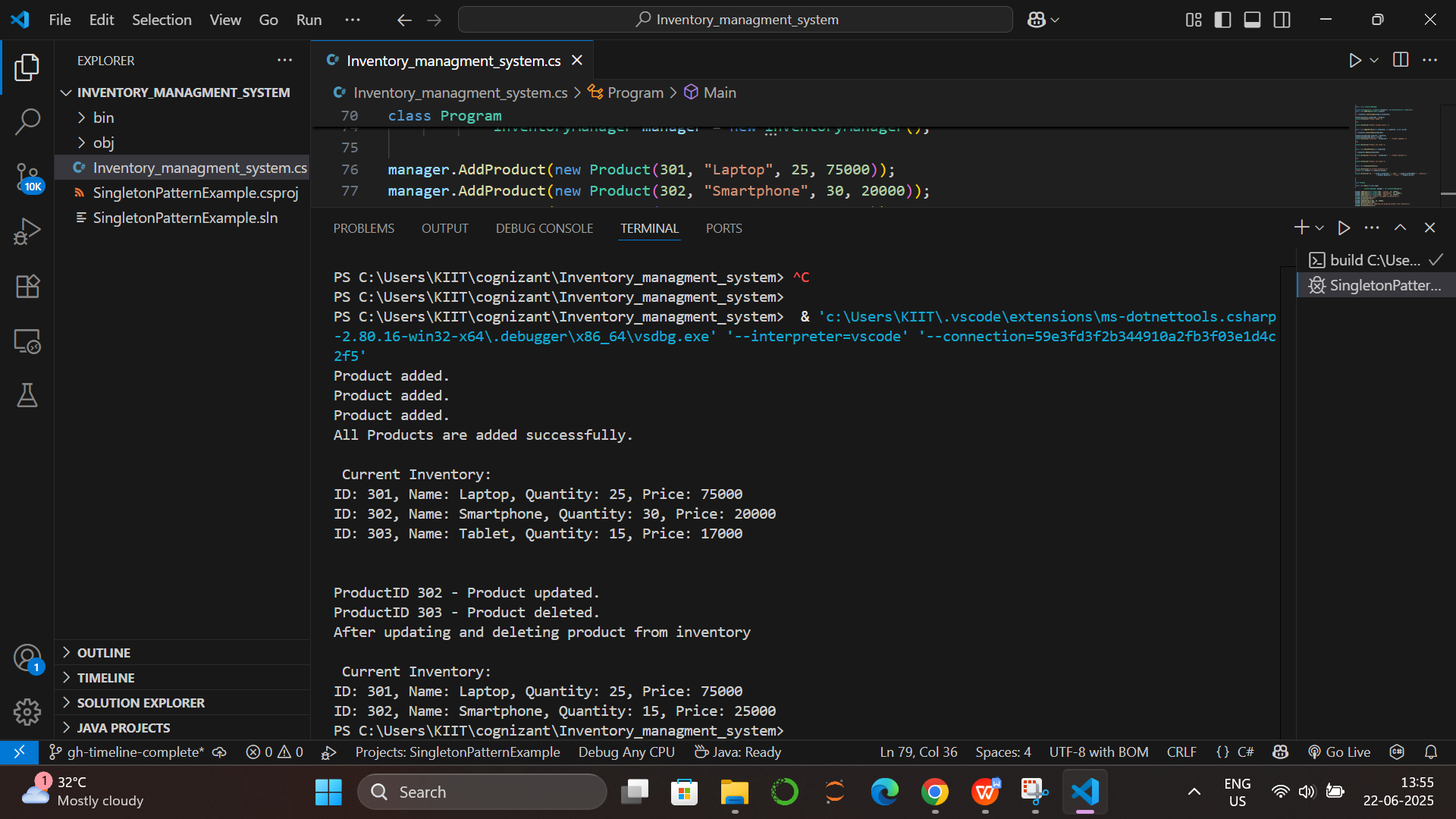
manager.DisplayInventory();

}

}

}

**Output:**



****Time Complexity Analysis:****

|  |  |  |
| --- | --- | --- |
| Operation | Average Time Complexity | Worst Time Complexity |
| Add Product | O(1) | O(n) |
| Update Product | O(1) | O(n) |
| Delete Product | O(1) | O(n) |
| Display All | O(n) | O(n) |

**Exercise 3: Sorting Customer Orders**

**Sorting algorithms Explanation**

1. Bubble Sort

Working:

Bubble Sort repeatedly compares adjacent elements and swaps them if they are in the wrong order. With each pass, the largest unsorted element "bubbles up" to its correct position at the end of the list.

Time Complexity:

Best Case: O(n) – when the list is already sorted

Average Case: O(n²)

Worst Case: O(n²)

Use Case:

Bubble Sort is mainly used for educational purposes. It is inefficient for large datasets due to its quadratic time complexity.

**Code**

using System;

using System.Collections.Generic;

namespace CustomerOrderSorting

{

public class Order

{

public int orderId;

public string customerName;

public double totalPrice;

public Order(int orderId, string customerName, double totalPrice)

{

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

public void Print()

{

Console.WriteLine("OrderID: " + orderId + ", Customer: " + customerName + ", TotalPrice: " + totalPrice);

}

}

class Program

{

public static void BubbleSort(List<Order> orders)

{

int n = orders.Count;

for (int i = 0; i < n - 1; i++)

{

for (int j = 0; j < n - i - 1; j++)

{

if (orders[j].totalPrice > orders[j + 1].totalPrice)

{

var temp = orders[j];

orders[j] = orders[j + 1];

orders[j + 1] = temp;

}

}

}

}

public static void QuickSort(List<Order> orders, int low, int high)

{

if (low < high)

{

int pi = Partition(orders, low, high);

QuickSort(orders, low, pi - 1);

QuickSort(orders, pi + 1, high);

}

}

public static int Partition(List<Order> orders, int low, int high)

{

double pivot = orders[high].totalPrice;

int i = low - 1;

for (int j = low; j < high; j++)

{

if (orders[j].totalPrice < pivot)

{

i++;

var temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

}

var temp1 = orders[i + 1];

orders[i + 1] = orders[high];

orders[high] = temp1;

return i + 1;

}

public static void PrintOrders(string title, List<Order> orders)

{

Console.WriteLine("\n" + title);

foreach (var order in orders)

{

order.Print();

}

}

static void Main(string[] args)

{

List<Order> orders = new List<Order>

{

new Order(301, "Ram", 5000),

new Order(302, "Shyam", 7000),

new Order(303, "Disha", 9300),

new Order(304, "Momo", 8300),

new Order(305, "Andres", 9000)

};

var bubbleSorted = new List<Order>(orders);

BubbleSort(bubbleSorted);

PrintOrders("Orders Sorted by Bubble Sort (Total Price Ascending):", bubbleSorted);

var quickSorted = new List<Order>(orders);

QuickSort(quickSorted, 0, quickSorted.Count - 1);

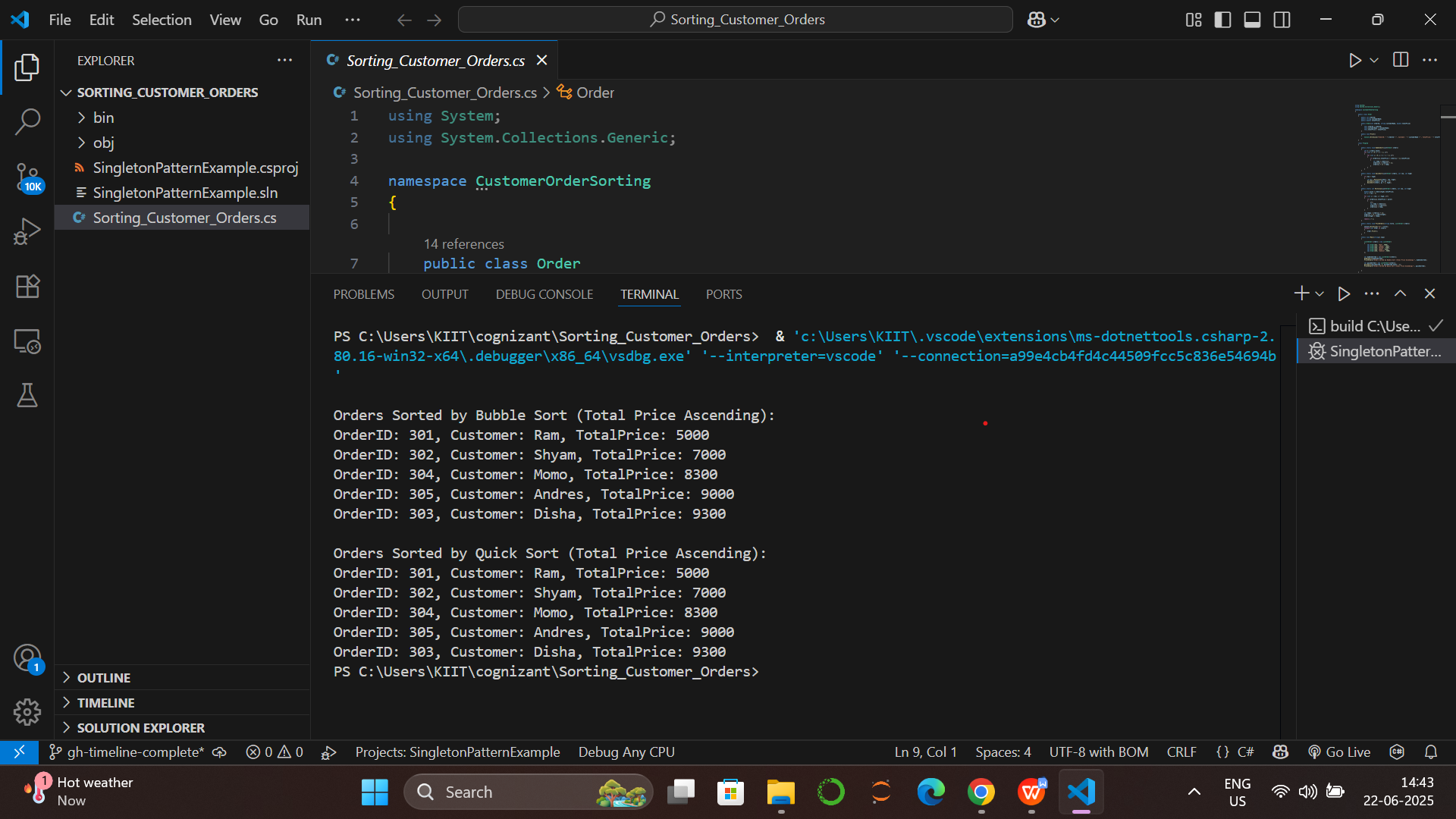
PrintOrders("Orders Sorted by Quick Sort (Total Price Ascending):", quickSorted);

}

}

}

**Output**



**Time Complexity Analysis**

Bubble Sort:

- Best: O(n)

- Average & Worst: O(n2)

Quick Sort:

- Best & Average: O(n log n)

- Worst: O(n2), but rare with good pivot selection

Quick Sort is preferred because:

- Much faster on large datasets

- More efficient use of memory

- Used in most real-world sorting libraries